프로그람개발문고

C++프로그람개발법

교육위원회 교육정보쎈터
주체99(2010)년

차　　례

머리말

위대한 령도자 **김정일**동지께서는 다음과 같이 지적하시였다.

《정보산업을 빨리 발전시키고 인민경제의 모든 부문을 정보화하여야 합니다.》

(《**김정일**선집》 제15권, 114페지)

위대한 령도자 **김정일**동지의 현명한 령도에 의하여 오늘 우리 나라에서는 인민경제의 정보화를 실현하기 위한 투쟁이 힘있게 벌어지고있다.

인민경제의 정보화를 실현하고 정보산업을 하루빨리 세계적수준에 끌어올리기 위해서는 정보공학 특히 프로그람기술을 빨리 발전시키고 능률적인 프로그람들을 적극 개발하여야 한다.

현실에서 요구되는 여러가지 응용프로그람들을 원만히 개발하기 위하여서는 프로그람언어와 프로그람작성법에 대한 깊은 지식을 소유하는것이 무엇보다도 중요하다.

프로그람작성에서 중요한 문제는 프로그람의 복잡성이다. 대규모프로그람은 사람들이 창조한것들중에서도 가장 복잡한 실체이므로 프로그람에서 오유를 범하기 쉽다. 프로그람오유는 많은 비용을 소비하게 하며 지어는 사람의 생명에도 위험을 준다.

객체지향프로그람작성법은 이와 같은 복잡성에 대응할수 있는 새로운 강력한 방도를 준다. 객체지향프로그람작성법의 목표는 프로그람을 더 명백하고도 믿음성이 있고 편리하게 개발하고 관리하는것이다.

객체지향언어들중에서 C++가 가장 널리 쓰이고있다. 지난 기간 C++의 표준은 진화단계에 있었으므로 번역프로그람제작자들마다 C++번역프로그람을 제각기 만들어냈다. 그러나 1997년 11월에 ANSI/ISO C++를 표준 C++의 최종안으로서 승인하였다. 표준 C++에는 표준형판서고(STL)를 비롯한 새로운 기능이 많이 추가되였다.

이 책에서는 C++프로그람언어에 의한 객체지향프로그람작성방법을 설명한다.

객체지향프로그람작성법은 Pascal, Basic, C와 같은 수속형언어의 프로그람작성자들에게 새로운 개념을 준다. 클라스, 계승, 다형성과 같은 개념은 객체지향프로그람작성법의 중심적인 개념들이다. 이 책에서는 먼저 이 개념들에 대한 일반적인 표상을 주고 점차적으로 객체와 클라스, 계승, 연산자의 재정의, 가상함수 등에 대하여 구체적으로 설명한다.

이 책에서는 C와 C++언어의 공통적인 수속적개념을 먼저 학습하고 그것을 리해한 다음 새로운 개념인 객체지향프로그람작성에로 넘어간다. 이 책의 목적의 하나가 객체지향프로그람작성을 될수록 빨리 시작하는데 있으므로 6장의 객체를 학습하기 전에 4장에서 그 기초인 수속을 학습한다. 또한 클라스는 문법적으로 구조체의 확장이고 C++를 리해하기 위한 중요한 개념이므로 클라스를 쉽게 리해할수 있도록 4장에서 구조체를 소개한다.

지적자와 같은 일부 개념들은 이전의 C서적들과 달리 책의 뒤부분에 주었다. 지적자는 객체지향프로그람작성법의 본질을 리해하는데 반드시 필요한것은 아니며 C와 C++를 배우는 사람들이 실수하기 쉬운 부분이다. 그러므로 이 책에서는 객체지향프로그람작성의 기본개념을 완전히 소유할 때까지 지적자를 론의하지 않는다.

또한 C++의 새 루틴들이 C의 일부 기능을 대신한다. 실례로 C++에서는 C의 입출력함수들을 드물게 사용하므로 설명하지 않는다. C의 상수와 마크로는 C++의 변경자와 inline함수가 대신한다.

이 책의 목적은 객체지향프로그람작성에 있으므로 객체지향프로그람작성과 관련되지 않은 특성들은 설명하지 않는다. 실례로 C의 비트연산자는 객체지향프로그람작성법을 학습하는데 필요하지 않으므로 설명하지 않는다.

이 책에서는 STL에 대한 새로운 장을 추가하였다.

현재 Windows조작체계에서는 Microsoft와 Borland에 의하여, Linux조작체계에서는 TrollTech를 비롯한 여러 회사들에 의하여 C++개발환경들이 개발되고있으며 이 책의 실례들은 표준 C++에 기초하고있으므로 Visual C++, C++ Builder, KDevelop를 비롯한 모든 C++번역프로그람들에 의하여 번역할수 있다.

이 책의 실례들은 조작탁방식프로그람이다. 조작탁방식프로그람은 번역프로그람환경내의 문자방식창문안에서 실행하거나 MS-DOS창문에서 직접 실행할수 있다.

이 책은 프로그람작성전문가, 대학생들을 위한 참고서이지만 이미 프로그람을 작성해본 경험이 없어도 사용할수 있으며 C언어를 알고있는것을 전제로 하지 않는다.

우리는 C++언어에 의한 객체지향프로그람작성법을 깊이 학습하여 자신의 프로그람개발능력을 더욱 높임으로써 인민경제의 현대화, 정보화를 실현하는데서 나서는 복잡한 과학기술적문제들을 능숙하게 풀어나갈수 있는 유능한 과학자로 준비되여야 한다.

제 1 장. 객체지향프로그람작성법의 기초

객체지향프로그람작성법의 두가지 기초개념은 객체와 클라스이다.

그러면 객체와 클라스는 무엇을 의미하는가? 또한 C++와 C언어사이의 관계는 어떠한가?

이 장에서는 이상과 같은 문제를 제기하고 객체지향프로그람작성법의 기본개념과 특성, C++와 C사이의 관계에 대하여 설명한다. 다음 장들에서 여기서 언급하는 개념들을 구체적으로 설명한다.

제 1 절. 객체지향프로그람작성법에 대한 기본개념

객체지향프로그람작성법(object-oriented programming)은 그 어떤 새로운 언어에 의한 프로그람작성방법인것이 아니라 프로그람설계에 대한 새로운 사고방법으로서 사람들이 객관세계를 파악하는 규칙 즉 인간의 사유방식에 맞게 객체를 서술할수 있게 하는 새로운 프로그람작성기술이다.

우리가 파악하려는 객관세계는 각이한 객체들로 이루어져있으며 그 객체들은 자기의 고유한 내부상태와 운동규칙을 가지고있다는것, 그리고 매 객체들은 서로 다른 객체들과 호상작용하고 련계를 맺으면서 변화발전한다는것을 포착하고 바로 그 객체를 단위로 하여 프로그람을 설계하는 객체중심의 프로그람작성수법이 객체지향수법이다.

어떤 사물현상에 대하여 고찰할 때 단순히 자료나 혹은 그것이 수행하는 어떤 기능 하나에 대하여서만이 아니라 그것을 다같이 종합적으로 생각하는것이 객체지향의 기본고찰방법이다.

객체지향에서 자료를 속성, 기능을 조작 또는 메쏘드라고 하고 이 두가지를 가지고 객체를 정의한다.

1. 객체지향에 대한 기본용어

1) 객체

객체지향체계는 객체를 중심으로 구성된다. 따라서 객체(object)는 객체지향수법의 핵으로 된다.

객체에 대한 개념과 해석에서 공통적으로 인식해야 할 점은 다음과 같다.

① 객체는 사람들이 연구하려는 임의의 사물현상이다.

객체는 형태를 가지는것도 있고 추상적인것도 있다. 즉

첫째로, 형태를 가지는 실체 례를 들면 비행기, 콤퓨터, 의자

둘째로, 사람이나 조직체가 일으키는 작용 례를 들면 교수, 청소

셋째로, 사건 즉 특정한 시간에 발생한 일

넷째로, 성능설명 즉 어떤 물질이나 물체의 속성 또는 기능설명 례를 들면 콤퓨터의 내부기능, 원자의 성질

② 객체는 자료(속성)와 메쏘드의 융합체이다.

모든 객체에는 각이한 내부상태(자료)와 운동규칙(그 자료에 대한 조작)이 있다. 객체는 바로 그 자료와 조작의 융합체이다.

③ 객체는 유일한 식별기능을 가지고있어야 한다.

객체는 속성과 행동방식의 두 측면에서 다른 객체와 구별된다.

따라서 객체를 소개할 때 체계에 의해 새 객체에게 유일한 객체표식부를 부여하며 그 표식부는 객체가 존재하는 기간 영구적으로 그 객체를 가리킨다.

서로 다른 객체는 서로 다른 표식부를 가진다.

또한 객체는 식별기능으로서 객체이름을 가지며 그것은 자료와 조작의 일체화된 모든것을 대표한다.

④ 객체는 반드시 적어도 하나이상의 어떤 무리에 포함되여 그 무리의 실체로 된다.

프로그람을 함수가 아니라 객체에 기초하여 고찰하면 간단히 프로그람을 설계할수 있다. 이것은 프로그람작성에서 객체와 현실세계의 객체들사이의 밀접한 일치로부터 흘러나온다.

프로그람작성에서 객체와 현실세계의 객체들사이의 일치는 자료와 함수를 결합한 결과이다. 수속형언어에서 프로그람구조와 모형화하고있는 항목사이에는 밀접한 일치가 존재하지 않는다.

2) 클라스

공통성을 가지고있는 객체 즉 자료구조와 조작에서 같은 규칙을 준수할수 있는 객체들의 집합을 클라스 또는 무리(class)라고 한다.

다시 말하여 클라스는 꼭같은 종류의 속성과 행동방식을 가지는 여러개의 객체를 종합하여 추상화한것이다.

클라스는 선조무리와 자손무리로 이루어지는 계층구조를 가진다.

객체는 반드시 어떤 무리에 속하여야 한다.

거의 모든 프로그람언어들은 기본자료형을 가지고있다. 실례로 자료형int는 옹근수를 의미하며 C++언어에 미리 정의되여있다. 프로그람에서 int형변수를 선언할수 있다.

```
int day;
int count;
```

마찬가지로 어떤 클라스의 객체들도 정의할수 있다. (그림 1-1) 클라스는 설계도 혹은 형판과 비슷하다.

클라스는 그의 객체들에 어떤 자료와 함수들이 포함되는가를 지정한다. 클라스정의는 객체를 창조하지 못한다. 마치도 int자료형의 존재자체가 어떤 변수를 창조하지 못하는것과 같다.

클라스는 류사한 많은 객체들의 서술이다. 례를 들면 승용차, 뻐스, 화물자동차는 운수수단클라스의 객체들이다.

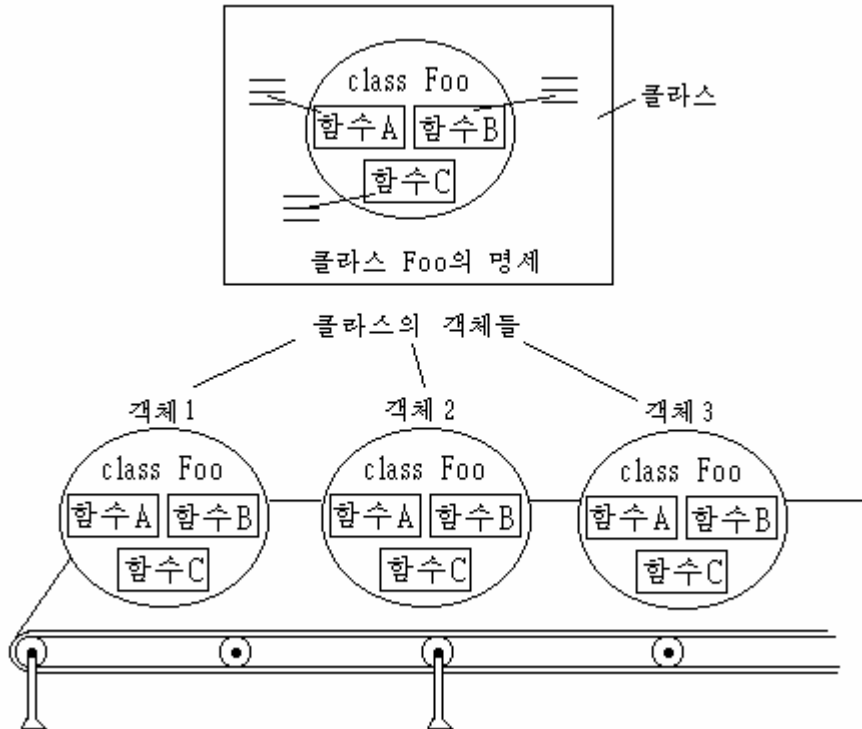



그림 1-1. 클라스와 객체

3) 속성

속성(attribute)은 여러개의 객체에 대하여 공통적인 성질을 나타내는 보조적인 정보이다. 속성은 클라스에 대하여 정의하는것으로서 그 클라스의 객체는 모두 같은 속성을 가진다.

객체는 매개의 속성에 고유한 값을 가지며 그 값은 행동방식(조작)과 함께 변한다.

4) 통보문과 메쏘드

객체들끼리 협조하여 어떤 작업을 하자면 서로 작업의뢰를 주고 받아야 한다. 일단 작업의뢰를 받은 객체는 그 작업의 조작순서대로 작업한다. 통보문(message)은 그 작업의뢰에 해당하고 메쏘드(method)는 작업의 조작순서에 대응하는 개념이다.

2. 객체지향체계의 특징

객체지향체계는 추상성, 밀봉성, 계승성, 다형성의 특징을 가진다.

① 추상성

추상성(abstraction)이란 주어진 문제가운데서 본질적인 성질과 특성만을 추려서 문제를 포착하는것을 말한다.

객체지향에서 현실세계의 사물현상을 객체로 보고 그것을 서술함에 있어서 필요한 정보만 강조하고 불필요한 정보를 생략하여 복잡한 현실세계를 간단명료하게 표시하는것이 바로 추상성이다.

이 추상성으로 하여 무리의 공통성을 찾아내고 무리에 대한 설명을 통일적으로함으로써 매 객체의 공통성에 대한 중복설명을 피할수 있게 한다.

② 밀봉성

객체지향에서는 자료와 그것을 가지고 할수 있는 조작(수속)까지 묶어서 하나의 객체로 정의하고 여기에 객체형이라는 자료형을 융합시킴으로써 이 객체의 자료에 대한 조작이 외부에서 임의로 가해지는것을 피하고 오직 자체객체안에서 정해진 행동규칙대로만 사용할수 있게 하는것이 객체의 밀봉성이다.

밀봉성으로 하여 객체밖에서는 오직 수속만을 호출할수 있고 이 수속을 통하여 간접적으로 객체안의 자료를 사용할수 있다.

그 모양이 마치도 자료가 껍데기안에 포위되여있는것처럼 보이므로 이것을 밀봉성(encapsulation) 또는 정보은폐(information hiding)라고 한다.

③ 계승성

현실세계는 계층구조를 이루는것이 많다.

계층구조에서 매 층은 하나의 클라스(무리)로 되며 여기서 보다 웃층의 클라스를 기초클라스(선조무리), 보다 아래층의 클라스를 파생클라스(자손무리)라고 한다.

보조클라스는 기초클라스가 가지고있는 모든것(속성과 조작)을 그대로 넘겨받는다. 바로 이와 같은 성질이 계승성이다.

계승성(inheritance)이란 계층관계에 기초하여 어떤 클라스의 속성과 조작이 다른 클라스와 공유하는것을 말한다. 이러한 관계를 두 클라스사이에 계승관계라고 한다.

계승원칙은 매개의 보조클라스가 그것을 파생시키는 클라스와 공통적인 특성을 공유해야 한다는것이다. 승용차, 화물자동차, 뻐스, 모터찌클은 모두 바퀴와 완충기를 가지고있으며 이것은 운수수단을 정의하는 특성들이다.

매개 보조클라스는 기초클라스의 다른 성원들이 공유하고있는 특성외에도 자기의 고유한 특성을 가지고있다. 례를 들면 뻐스는 사람들이 앉기 위한 좌석이 있고 화물자동차는 짐을 싣기 위한 적재함이 있다. 이 개념을 그림 1-2에 주었다. 그림에서 특성 A와 B는 기초클라스의 부분이며 모든 파생클라스들이 공유하지만 매개 파생클라스는 자기의 고유한 특성들도 가지고있다는것을 알수 있다.

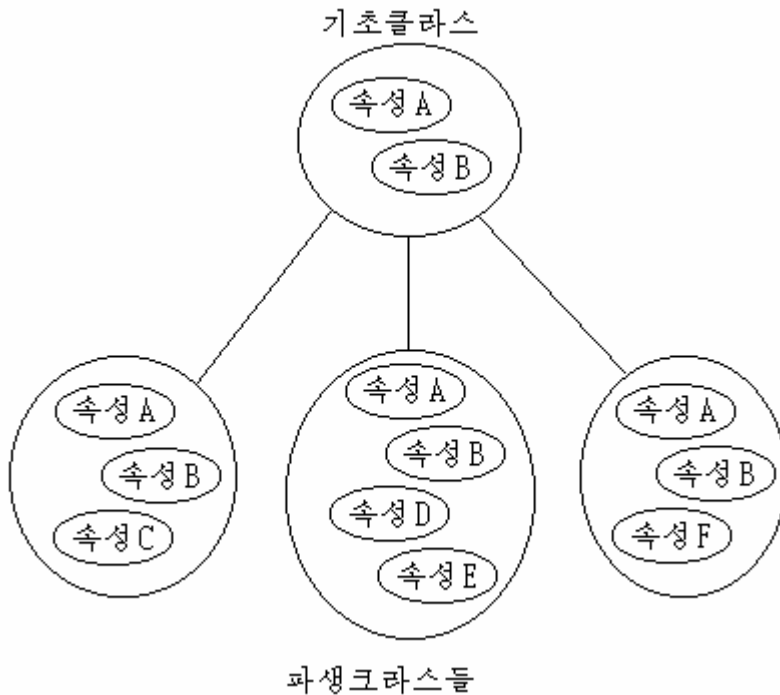



그림 1-2. 계승

마찬가지로 객체지향프로그람의 클라스도 보조클라스들로 분류할수 있다.

C++에서는 원시클라스를 기초클라스(base class)라고 부르며 다른 클라스들은 기초클라스의 특성을 공유하는것으로 정의할수 있지만 자체의 특성을 추가적으로 가지고있다. 이러한 클라스를 파생클라스(derived class)라고 한다.

객체와 클라스의 관계는 기초클라스와 파생클라스의 관계와 다르다.

객체는 콤퓨터의 기억기안에 존재하며 매 객체는 형판로서 사용되는 클라스의 구체적인 특성들을 가지고있다.

파생클라스에는 기초클라스들로부터 특성들이 계승되지만 그밖에도 자체의 새 특성들이 추가된다.

계승은 전통적인 수속형프로그람을 단순화하기 위하여 함수를 사용하는것과 같은것이다.

만일 수속형프로그람의 세개의 각이한 부분들이 거의 같은것을 수행한다는것을 발견하면 이 세개 부분들의 공통요소들을 추출하여 그것들을 하나의 함수에 넣는다. 프로그람의 세개 부분은 공통동작을 수행하기 위하여 그 함수를 호출한다면 함수를 자기의 개별적인 처리처럼 실행할수 있다. 마찬가지로 기초클라스는 파생클라스들에 공통인 요소들을 포함한다. 함수를 수속형프로그람에서 실행할 때 계승은 객체지향프로그람작성을 간단화하며 문제요소들사이의 관계를 명백하게 해준다.

④ 다형성

다형성(polymorphism)은 현실에 존재하는 행동의 원활성과 반복사용을 반영한것

으로써 같은 조작이라고 하여도 서로 다른 클라스에 적용하면 서로 다른 결과를 얻게 하는 성질이다.

실례로 수자형시계나 상사형시계에 꼭같이 현재시간을 표시하라고 지시하면 수자형시계는 시간을 수자로 표시하고 상사형시계는 시, 분, 초로 표시한다.

이것은 서로 다른 방법으로 현재시간을 표시하는 조작을 실현할수 있다는것을 의미한다.

어떤 무리에 대한 고유한 조작의 실현을 메쏘드라고 한다.

객체지향에서의 조작은 다형성을 가지고있으므로 하나의 조작에 대하여 그것을 실현하는 방법은 하나이상일수 있다.

⑤ 재리용성

일단 클라스를 정의하고 오유를 수정하였다면 다른 프로그람들에서 클라스를 사용할수 있다. 이것을 재리용성(reusuablity)이라고 한다.

이것은 수속형언어의 함수서고가 각이한 프로그람들과 결합할수 있는것과 비슷하다.

그러나 객체지향프로그람작성에서 계승은 재리용성을 제공해준다.

작성자는 현존클라스를 그대로 사용할수도 있고 그것을 변경하지 않고도 거기에 새로운 특성과 기능을 추가할수도 있다. 이것은 현존클라스로부터 새 클라스를 파생시키는 방법으로 진행된다.

새 클라스는 이전 클라스의 기능들을 계승하며 또한 그 자체만 가지고있는 새 특성이 추가된다.

실례로 Windows나 다른 GUI에서 사용하는것과 같은 안내체계를 창조하는 클라스를 쓸수 있다. 이 클라스는 잘 동작하므로 그것을 변경하려고 하지 않지만 일부 안내입구들을 가능 혹은 불가능하게 할수 있는 능력을 추가하려고 한다. 그러자면 현존클라스의 능력을 모두 계승하며 안내입구의 가능성을 추가한 새 클라스를 간단히 창조한다.

현존 쏘프트웨어를 재리용할수 있는것은 객체지향프로그람작성법의 중요한 우점이다.

⑥ 새 자료형의 창조

객체는 프로그람작성자에게 새 자료형을 창조하는데 편리한 수단을 제공해준다.

실례로 프로그람에서 x, y자리표나 위도, 경도와 같은 2차원위치를 취급한다고 가정하자. 이때 표준연산자로 위치값들에 대한 연산을 하면 편리하다. 즉

```
position1 = position2 + origin;
```

변수 position1, position2, origin은 각각 독립적인 수량을 표시한다.

위치를 나타내는 클라스를 창조하고 position1, position2, origin을 이 클라스의 객체로 선언함으로써 새 자료형을 효과적으로 사용할수 있다.

C++는 이러한 방법으로 새 자료형을 창조할수 있게 해준다.

제 2 절. C++와 C언어

C++는 C로부터 파생된 언어이다. 즉 C++는 C의 상위모임이다. 따라서 C에서 정확히 동작하는 명령문은 C++에서도 정확히 동작한다. 그러나 반대로는 동작하지 않는다.

C++를 만들 때 C에 추가한 클라스, 객체와 같은 가장 중요한 요소들은 객체지향프로그람작성과 관련되여있다. C++는 처음에 클라스를 가진 C라고 불렀다. 그러나 C++는 입출력과 설명문 등 새로운 특성들을 수많이 가지고있다. 그림 1-3은 C와 C++사이의 관계를 보여준다.

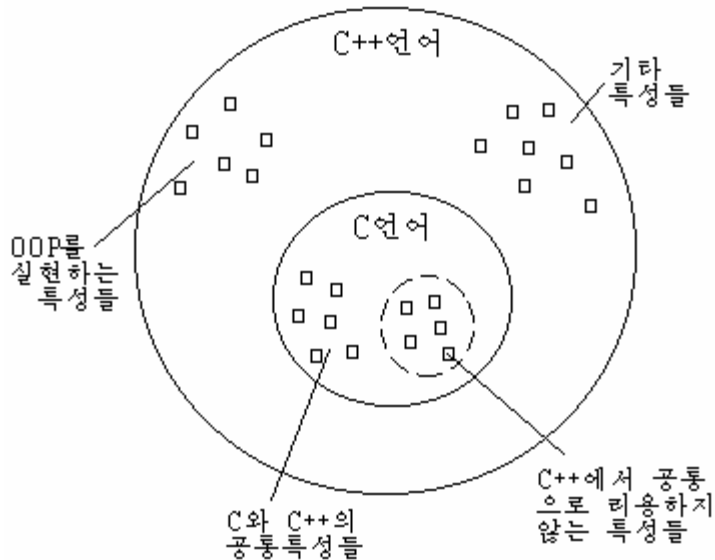



그림 1-3. C와 C++사이의 관계

사실상 C와 C++사이의 실천적차이는 크다. C로 쓴 프로그람을 C++로 쓴 프로그람에서 사용할수 있다. C++프로그람작성자는 C++의 새로운 특성들과 함께 전통적인 C의 특성들도 사용할수 있다.

C를 이미 알고있다면 C++프로그람작성에서 도움이 되지만 C++의 대부분은 새로운것이다.

요 약

객체지향프로그람작성법은 프로그람을 구성하는 하나의 방법이다. 요점은 코드작성이 아니라 프로그람을 설계하는 방법에 있다. 특히 객체지향프로그람들은 객체들로

구성되고 객체는 자료와 그 자료에 대하여 동작하는 함수들을 둘다 포함한다. 클라스는 많은 객체들의 설계도 혹은 형판이다.

클라스는 공통성을 가지는 객체 즉 자료구조와 조작에서 같은 규칙을 준수할수 있는 객체들의 집합이다.

객체지향체계는 추상성과 밀봉성, 계승성, 다형성의 특징을 가진다.

추상성이란 주어진 문제가운데서 본질적인 성질과 특성만을 추려서 문제를 포착하는것을 말한다.

자료와 자료에 대한 조작(수속)까지 묶어서 하나의 객체로 정의하고 여기에 객체형이라는 자료형을 융합시킴으로써 객체의 자료에 대한 조작이 외부에서 임의로 가해지는것을 피하고 오직 객체안에서 정해진 행동규칙대로만 사용할수 있게 하는것이 객체의 밀봉성이다.

계승성이란 계층관계에 기초하여 어떤 클라스의 속성과 조작이 다른 클라스와 공유하는것을 말한다. 이러한 관계를 두 클라스사이의 계승관계라고 한다.

다형성은 현실에 존재하는 행동의 원활성과 반복사용을 반영한것으로서 같은 조작이라고 하여도 서로 다른 클라스에 적용하면 서로 다른 결과를 얻게 하는 성질이다.

객체지향프로그람작성에서 계승은 재리용성을 제공해준다.

일단 클라스를 정의하고 오유를 수정하였다면 다른 프로그람들에서 클라스를 사용할수 있다. 이것을 재리용성이라고 한다.

객체는 프로그람작성자에게 새로운 자료형을 창조하는데 편리한 수단을 제공해준다.

C++는 C의 웃준위모임이다. C++는 C언어에 객체지향프로그람작성방법을 실현할 능력을 제공한다. C의 일부 특성들은 C++에서 유효하지만 드물게 쓰이며 일부 다른 특성들은 자주 쓰인다. 한마디로 말하여 C++는 C와 완전히 다른 언어이다.

문 제

1. Pascal, Basic, C, C++는 각각 어떤 종류의 언어인가?

2. 운수수단이라는것은

ㄱ) 성원함수인가?

ㄴ) 클라스인가?

ㄷ) 연산자인가?

ㄹ) 자료항목인가?

3. 객체의 두가지 중요한 구성요소는 무엇인가?

4. C++언어에서는 클라스에 포함된 함수를

ㄱ) 성원함수라고 하는가?

ㄴ) 연산자라고 하는가?

ㄷ) 클라스함수라고 하는가?

ㄹ) 메쏘드라고 하는가?

5. 자료나 함수들에 대한 허용되지 않은 호출로부터의 보호를 무엇이라고 하는가?

6. 객체지향언어를 사용하는 리유가 다음과 같다고 말할수 있는가?

ㄱ) 자체로 자료형을 정의할수 있기때문이다.

ㄴ) 프로그람명령문이 수속형언어보다 단순하기때문이다.

ㄷ) 객체지향프로그람이 자체의 오유수정방법을 알려주기때문이다.

ㄹ) 객체지향프로그람작성법을 개념화하기 쉽기때문이다.

7. 무엇이 현실세계의 실체들을 함수보다 더 엄밀하게 모형화하는가?

8. C++프로그람은 코드작성의 세부를 제외하면 C프로그람과 비슷하다고 말할수 있는가?

9. 자료와 함수를 하나로 융합하는것을 무엇이라고 하는가?

10. 언어가 새로운 자료형을 창조할 능력을 가질 때 다음과 같이 말할수 있는가?

ㄱ) 밀봉할수 있다.

ㄴ) 재정의할수 있다.

ㄷ) 확장할수 있다.

11. 임의의 두 행의 코드를 보고 프로그람을 C로 썼다, C++로 썼다고 말할수 있는가?

12. 함수 혹은 연산자가 각이한 자료형에 대하여 서로 다른 방법으로 조작하는 능력을 무엇이라고 하는가?

13. 새로 정의한 자료형에 대하여 독특한 방법으로 조작하는 보통의 C++연산자는

ㄱ) 밀봉된다.

ㄴ) 분류된다.

ㄷ) 재정의된다고 말할수 있는가?

제 2 장. C++프로그람의 기본구조

이 장에서는 우선 C++언어의 세가지 기본요소 즉 프로그람의 기본구조와 변수, cout와 cin에 의한 입출력에 대하여 설명한다. 또한 설명문, 산수연산자, 대입과 증가연산자, 자료변환, 서고함수와 같은 언어의 몇가지 기능을 설명한다.

제 1 절. C++프로그람의 기본구조

실례 2-1은 가장 간단한 C++프로그람을 보여준다. 이 프로그람은 화면에 한개의 문장을 출력한다.

(실례 2-1) C++프로그람의 기본구조

```
#include <iostream>
using namespace std;
int main()
{
  cout << "안녕하십니까!\n";
  return 0;
}
```

이 프로그람은 비록 크지 않지만 C++프로그람의 기본구조를 보여준다.

1. 함수

함수(function)는 C++의 기본구성요소의 하나이다. 실례 2-1은 main()이라는 하나의 함수로 이루어진다. 프로그람에서 처음의 두 행 즉 #include와 using으로 시작하는 행들은 함수부분이 아니다.

함수가 클라스의 한부분일 때 그것을 성원함수(member function)라고 한다. 그러나 함수는 클라스와 독립적으로 존재할수도 있다. 여기서는 main()과 같이 독립적으로 존재하는 함수들을 고찰한다.

1) 함수이름

main의 뒤에 오는 괄호 ()는 함수의 독특한 모양을 보여준다. 괄호가 없을 때 번역프로그람은 main을 변수 혹은 다른 프로그람요소로 간주한다. 함수를 서술할 때에는 C++에 제정된 문법을 따라야 한다. 즉 함수이름뒤에 괄호를 붙여야 한다. 괄호가 항상 비여있는것은 아니다.

괄호는 함수의 인수(argument) 즉 그 함수를 호출하는 프로그람으로부터 넘어오는 값들을 보관하는데 사용된다.

함수이름앞에 놓인 예약어 int는 이 함수가 int형의 돌림값(return value)을 가진다는것을 의미한다.

2) 대괄호와 함수본체

함수본체(body)는 대괄호 {}안에 들어있다. 대괄호는 다른 언어들에서 BEGIN과 END예약어와 같은 역할을 한다. 대괄호는 프로그람명령문들의 한개 블로크를 둘러싸거나 경계를 결정한다.

모든 함수는 함수본체를 둘러싸는 한 쌍의 대괄호를 가진다. 실례 2-1에서는 함수본체안에 오직 두개 명령문 즉 cout로 시작하는 행과 return으로 시작하는 행이 있다. 함수본체는 여러개의 명령문들로 이루어진다.

3) 항상 main()으로부터 실행한다.

C++프로그람은 main()함수의 선두에 있는 첫 명령문으로부터 실행된다. 프로그람은 함수와 클라스, 그밖의 프로그람요소들로 이루어질수 있으나 실행시에 조종은 항상 main()에로 넘어온다. 프로그람에 main()함수가 없으면 오유를 통보한다.

대부분의 C++프로그람들에서 main()은 여러 객체들안에 있는 성원함수들을 호출한다. main()함수는 또한 다른 독립적인 함수들의 호출도 포함할수 있다. 이것을 그림 2-1에 주었다.

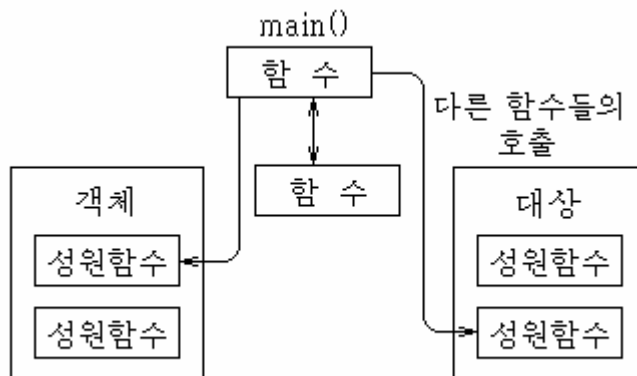



그림 2-1. 객체, 함수와 main()

2. 프로그람명령문

프로그람명령문(statement)은 C++프로그람의 기본단위이다. 실례 2-1에는 두개의 명령문이 있다. 즉 행

```
cout << "안녕하십니까!\n";
```

과 return명령문

```
return 0;
```

첫 명령문은 콤퓨터에 인용문을 표시하게 한다. 대부분의 명령문들은 콤퓨터에게 어떤 일을 수행하게 한다. 이 점에서 C++의 명령문은 다른 언어의 명령문과 같다.

반두점은 명령문의 끝을 의미한다. 반두점은 문법적으로 정해진 부분이지만 잊어버리기 쉽다. 일부 언어에서 명령문의 끝은 행끝으로 되지만 C++에서는 명령문의 끝

에 반두점을 쓰지 않으면 번역프로그람이 오유를 통보한다.

함수본체의 마지막 명령문은 return 0;이다. 이 명령문은 함수를 호출한 측에 값 0을 돌려준다. 이 경우에는 조작체계 혹은 번역프로그람에 넘겨준다. 이전의 C++판에서는 void형의 돌림값을 돌려줄 때 return명령문을 생략할수 있었지만 표준 C++에서는 이것이 옳다고 말할수 없다.

3. 공백

공백은 C++번역프로그람에서 중요하지 않다. 사실상 번역프로그람은 공백을 거의 무시한다.

공백(whitespace)은 공백문자(space), 복귀문자(carriage return), 행바꾸기문자(linefeed), 타브문자(tab), 수직타브문자(vertical tab), 종이바꾸기문자(formfeed)로서 정의한다. 이 공백문자들은 번역프로그람이 볼수 없다. 한 행에는 구분기호에 의해 구분되는 여러개의 명령문들을 놓을수 있으며 한 명령문을 두개이상의 행에 놓을수 있다.

따라서 실례 2-1의 프로그람을 다음과 같이 쓸수 있다.

```
#include <iostream>
using namespace std;
int main() { cout << "안녕하십니까!\n";   return 0; }
```

이것은 표준이 아니므로 읽기 힘들지만 번역은 정확히 수행된다.

공백을 쓸 때 지켜야 할 몇가지 규칙이 있다. 우선 프로그람의 첫행은 #include라는 앞처리지령으로 시작해야 하며 한행에 써야 한다. 또한 문자렬상수

```
"안녕하십니까!\n"
```

는 여러행에 갈라 쓸수 없다. 긴 문자렬상수를 쓸 때에는 행끝에 역사선(\)을 넣거나 두개의 문자렬을 제각기 2중인용표(")에 넣어 갈라서 쓸수 있다.

4. cout를 사용한 출력

명령문

```
cout << "안녕하십니까!\n";
```

는 인용표안에 넣은 문장을 화면에 표시한다. 이 명령문을 완전히 리해하려면 객체, 연산자재정의, 그밖의 개념들을 알아야 한다.

식별자 cout("c out")는 실제상 하나의 객체이다. 이것은 C++에서 표준출력스트림에 이미 정의되여있다. 스트림(stream)은 자료의 흐름을 참고하는 추상적인 개념이다. 표준출력스트림은 보통 영상표시장치에로 흐른다.

연산자 << 는 삽입(insert) 혹은 출력(put to)연산자라고 한다. 이 연산자는 오른쪽에 있는 변수의 내용이 그 왼쪽에 있는 객체에로 흐르게 한다. 실례 2-1에서 이것은 문자렬상수 "안녕하십니까!\n"이 영상표시장치를 의미하는 cout로 가게 한다.

그림 2-2는 cout와 삽입연산자 <<의 사용결과를 보여준다.

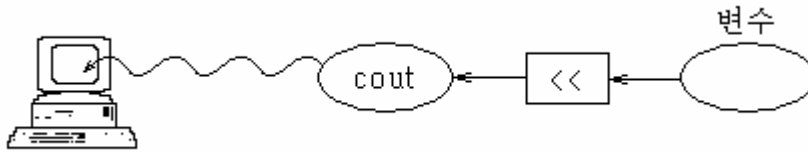



그림 2-2. cout에 의한 출력

- 문자렬상수

인용표안의 문장 "안녕하십니까!\n"는 문자렬상수의 실례이다. 변수와는 달리 상수에는 프로그람을 실행할 때 새 값을 줄수 없다. 상수의 값은 프로그람을 쓸 때 설정되며 프로그람을 실행할 때까지 그대로 보존된다.

일반적으로 문자렬을 두가지 방법으로 표시한다. 즉 문자렬을 문자들의 배렬로 표시하거나 어떤 클라스의 객체로 표시할수 있다.

문자렬상수의 끝에 있는 문자 '\n'은 확장문자인 행바꾸기문자이다. 이 문자는 다음의 본문을 새 행에 출력하게 한다. 행바꾸기문자는 프로그람의 실행이 끝난 후에 "Press any key to continue."라는 문장이 다음 행에 출력되게 한다.

제 2 절. 지령

실례 2-1의 처음 두 행은 지령(directive)이다. 하나는 앞처리(preprocessor)지령이고 다른 하나는 using지령이다. 이 지령들은 C++언어의 기본부분은 아니지만 필요하다.

1. 앞처리지령

실례 2-1의 첫행 #include <iostream>은 프로그람명령문이 아니다. 이 행은 함수본체의 한 부분도 아니고 프로그람명령문처럼 반두점으로 끝나지도 않는다. 그 대신에 #기호로 시작한다. 이것을 앞처리지령이라고 한다. 프로그람명령문은 콤퓨터에게 어떤 일을 시키는 명령이다. 앞처리프로그람이라고 하는 번역프로그람의 한 부분은 실제로 번역을 시작하기 전에 앞처리지령들을 처리한다.

앞처리지령 #include는 번역프로그람이 원천파일안에 다른 파일을 삽입하게 한다. 실례로 #include지령은 그 뒤에 지적된 파일의 내용으로 교체된다. #include지령을 사용하여 원천파일에 다른 파일을 삽입하는것은 본문의 한 부분을 문서처리프로그람에 읽어들인 문서에 붙이기(paste)하는것과 같다.

#include는 앞처리지령의 하나이며 앞처리지령은 처음의 #기호에 의하여 식별된다.

일반적으로 앞처리지령의 사용법은 C++와 C에서 같지 않다. #include에 의해 포함되는 형파일(type file)을 머리부파일(header file, include file)이라고 한다.

2. 머리부파일

실례 2-1에서 앞처리지령 #include는 번역프로그람에게 번역하기 전에 실례의 원천파일에 다른 원천파일 IOSTREAM을 추가하게 한다.

그러면 왜 IOSTREAM을 추가하는가?

IOSTREAM은 머리부파일의 하나의 실례이다. 이 파일은 기본입출력조작과 관련되여있으며 cout식별자와 <<연산자가 요구하는 선언들을 포함하고있다. 만일 이러한 선언이 없으면 번역프로그람은 cout와 <<를 인식할수 없으며 오유를 통보한다.

머리부파일은 많다. 새로 갱신된 표준 C++머리부파일들은 파일확장자를 가지지 않는다. 그러나 이전 판의 머리부파일들은 확장자를 가진다.

IOSTREAM의 내용을 보려면 번역프로그람용 Include등록부를 탐색하여 편집창문에 원천파일로서 표시하면 된다. 혹은 WordPad나 NotePad를 사용하여 볼수도 있다.

3. using지령

C++프로그람은 여러가지 이름공간으로 나눌수 있다. 이름공간(name space)은 일정한 이름들을 인식하는 프로그람의 한 부분이다. 즉 어떤 이름공간의 안에서 정의한 이름은 그 이름공간의 밖에서는 알려지지 않은 이름으로 된다. 지령 using namespace std;은 뒤에 오는 모든 프로그람명령문들이 std이름공간안에 있다는것을 알려준다.

cout와 같은 몇가지 프로그람요소들은 std이름공간안에서 선언된다. using지령을 사용하지 않으려면 실례 2-1을 다음과 같이 써야 한다.

```
std::cout << "안녕하십니까!\n";
```

프로그람안에 매번 std::를 추가하지 않으려면 using지령을 써야 한다.

제 3 절. 설명문

설명문(comment)은 프로그람에서 중요한 부분의 하나이다. 설명문은 프로그람작성자를 방조하며 누구나 원천파일을 읽고 리해하게 한다. 번역프로그람은 설명문을 무시하므로 실행프로그람의 파일크기에 영향을 주지 않으며 실행시간도 늘어나지 않는다.

1. C++의 설명문

실례 2-2는 실례 2-1의 프로그람에 설명문을 붙인것이다.

(실례 2-2) C++의 설명문

```
#include <iostream>                // 앞처리지령
using namespace std;               // using지령
int main()                         // main()함수
{                                  // 함수본체의 시작
```

```
    cout << "안녕하십니까!\n";        // 명령문
    return 0;                         // 명령문
}                                     // 함수본체의 끝
```

설명문은 두개의 사선기호(//)로 시작되고 행끝에서 끝난다. 설명문은 행의 선두로부터 시작할수도 있고 프로그람명령문과 한행에 놓일수 있다

설명문은 다음과 같은 경우에 사용한다.

- 프로그람의 동작에 대한 구체적인 설명이 필요할 때
- 후에 프로그람의 구체적인 조작을 쉽게 리해하기 위하여

2. C의 설명문

C++에서 사용할수 있는 두번째 형식의 설명문은 다음과 같다.

```
/* 이것은 낡은 형식의 설명문이다. */
```

이 형식의 설명문은 /*로 시작하여 */로 끝난다. 이 설명문은 입력하는데 품이 들고 많은 행을 차지하므로 C++에서는 대체로 사용하지 않는다. 그러나 두개의 설명기호를 가지고 여러행의 설명문을 쓸 때에는 편리하다. 례를 들면

```
/* 이것은 여러행 설명문이다. */
```

여러행의 설명문을 쓸 때에는 행마다 //를 쓰는것보다 우와 같은 설명문을 쓰는것이 더 좋다.

또한 프로그람행의 본문중에 어디에나 /* */설명문을 넣을수 있다.

```
Func1()
{ /* 빈 함수본체 */ }
```

이 경우에 //형의 설명문을 쓰면 번역프로그람이 닫긴 괄호를 인식할수 없다.

제 4 절. 옹근수변수

변수(variable)는 언어의 가장 기초적인 부분이다. 변수는 기호이름을 가지며 여러가지 값을 가질수 있다. 변수들은 콤퓨터기억기안의 특정한 위치에 배치된다. 변수에 값이 주어지면 그 값은 사실상 변수에 할당된 기억공간에 보관된다. 대부분의 언어들에서는 옹근수형, 류동소수점수형, 문자형의 일반적인 변수형을 사용하며 변수에 대한 개념이 서로 비슷하다.

옹근수변수는 1, 30000, -27과 같은 옹근수(integer)를 표시한다. 옹근수는 연필 1자루, 책 99권 등 객체의 수량을 세는데 쓰인다. 옹근수는 류동소수점수와 달리 소수부를 가지지 않는다.

1. 옹근수변수의 정의

옹근수변수의 류형에는 여러가지가 있으며 보통 int형이 많이 쓰인다. int형이 차지하는 기억기의 크기는 체계에 의존한다. Windows 98과 같은 32bit체계에서 int는

4byte기억공간(32bit)을 차지하고 -2,147,483,648∼2,147,483,647의 값범위를 가진다. 그림 2-3은 기억기안에서의 옹근수변수를 보여준다.

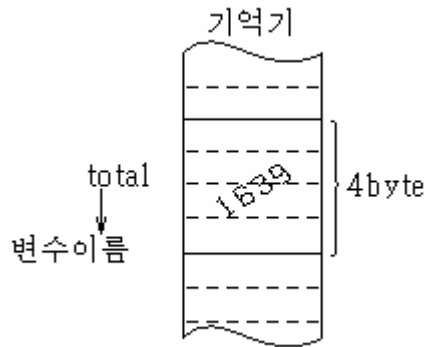



그림 2-3. 기억기안에서의 옹근수변수

현재의 Windows체계들에서 int는 4byte를 차지하지만 MS-DOS와 3.1이하의 Windows에서는 2byte를 차지한다. 각종 형들이 차지하는 기억범위는 머리부파일 LIMITS에 들어있다. 이것을 방조체계를 통하여 볼수 있다.

실례 2-3에서는 int형의 변수를 여러개 정의하고 사용한다.

(실례 2-3) 옹근수변수

```
#include <iostream>
using namespace std;
int main()
{
   int var1;                       // var1을 정의한다.
   int var2;                       // var2을 정의한다.
   var1 = 20;                      // var1에 값을 대입한다.
   var2 = var1 + 10;               // var2에 값을 대입한다.
   cout << "var1 + 10 =";          // 본문을 출력한다.
   cout << var2 << endl;           // var2의 값을 출력한다.
   return 0;
}
```

명령문

```
int var1;
int var2;
```

은 두개의 옹근수변수 var1과 var2를 정의한다. 예약어 int는 변수의 형을 의미한다. 선언이라고 부르는 이 명령문들은 다른 프로그람명령문처럼 반두점으로 끝나야 한다.

변수는 사용하기 전에 선언하여야 한다. C에서는 첫 실행명령문앞에서 변수를 선언해야 하지만 C++에서는 변수선언을 프로그람안의 어디에나 놓을수 있다. 그러나 공통으로 사용하는 변수들은 프로그람의 선두에 배치하는것이 리해하는데 편리하다.

2. 선언과 정의

변수에 대하여 선언과 정의의 차이를 고찰해보자.

선언(declaration)은 변수의 이름(var1 등)을 프로그람에 받아들이고 그의 형(int 등)을 지정한다. 그러나 선언이 변수용기억기까지 설정한다면 그것을 정의(definitoin)라고도 한다.

실례 2-3의 명령문

```
int var1;
int var2;
```

은 var1과 var2용 기억기도 설정하므로 정의이다. 대체로 선언은 정의와 련관되여있지만 후에 정의가 아닌 선언에 대하여 고찰한다.

3. 변수이름

실례 2-3의 프로그람은 var1과 var2라는 변수를 사용한다.

변수들에 주어진 이름을 식별자(identifier)라고 부른다. 그러면 식별자를 쓰는 규칙은 무엇인가?

식별자는 영어대소문자와 0～9의 수자를 사용하여 쓸수 있다. 또한 밑줄(_)을 사용할수 있다. 식별자의 첫 문자는 영문자 혹은 밑줄이여야 한다. Visual C++번역프로그람은 식별자의 첫 247문자, C++ Builder는 250문자만 인식한다. 번역프로그람은 대소문자를 구별한다. Var와 var, VAR는 서로 다르다.

C++예약어를 변수이름으로 사용할수 없다. 예약어(keyword)는 int, class, if, while과 같이 특별한 의미를 가지는것으로 미리 정의된 단어이다.

대부분의 C++프로그람작성자들은 변수이름으로써 모두 소문자를 사용한다. 일부 프로그람작성자들은 IntVar혹은 dataCount와 같이 대소문자를 섞어서 사용한다. 어느 방법을 사용하든 한개 프로그람에서는 한가지 방법을 일관하게 사용하여야 한다. 모두 대문자로 쓴 이름은 대체로 상수용으로 예약되여있다.

이것은 클라스나 함수와 같은 다른 프로그람요소들의 이름을 짓는데에도 적용할수 있다.

4. 대입명령문

명령문

```
var1 = 20;
var2 = var1 + 10;
```

은 두개 변수에 값을 대입한다. 같기기호 =는 오른변값을 왼쪽변수에 대입하게 한다.

C++의 =는 Pascal의 :=나 BASIC의 =에 대응된다. 첫 행에서 var1에는 값 20이 대입된다.

5. 옹근수상수

수 20은 옹근수상수(integer constant)이다. 상수는 프로그람을 실행하는 전과정에

변하지 않는다. 옹근수상수는 수자들로 이루어진다. 옹근수상수안에 소수점이 없어야 하며 옹근수들의 범위안에 있어야 한다.

프로그람의 둘째 행에서 +기호는 var1의 값과 10을 더하여 결과를 var2에 대입한다. 여기서 10은 다른 상수이다.

6. 다른 출력방법

명령문

```
cout << "var1 + 10 =";
```

은 문자렬상수를 화면에 표시한다. 다음 명령문

```
cout << var2 << endl;
```

은 변수 var2의 값을 화면에 표시한다. 프로그람의 출력은 다음과 같다.

```
var1 + 10 = 30
```

cout와 <<연산자는 옹근수와 문자렬을 서로 다르게 취급하는 방법을 보여준다. 만일 cout에 문자렬을 보내면 문자렬을 본문으로 출력하고 옹근수를 보내면 옹근수를 수값으로 출력한다.

두개의 cout명령문의 출력은 출력화면의 한행에 나타난다. 행바꾸기기호는 저절로 삽입되지 않는다. 새로운 행에서 시작하려면 자체로 행을 바꾸어야 한다. 행바꾸기에는 확장문자 '\n'를 사용한다. 또한 조작자를 사용할수 있다.

- endl조작자

실례 2-3에서 마지막 cout명령문은 endl로 끝난다. endl은 스트림에 행바꾸기를 삽입하여 그 뒤에 출력하는 본문이 새로운 행에 놓이게 한다. 이것은 '\n'를 보내는것과 결과는 같지만 더 명백한 방법이다. 조작자(manipulator)는 여러가지 방법으로 출력을 변경시키는 출력스트림에 대한 명령이다. 또한 endl은 '\n'과는 달리 출력완충기에 들어있는 자료를 한번에 모두 내보내게 한다.

7. 다른 옹근수형

옹근수형에는 int형외에 long형과 short형이 있다. 엄격히 말한다면 char형도 옹근수형이지만 문자형으로 따로 구분한다. int형의 크기는 체계에 의존하지만 long과 short는 사용하는 체계에 관계없이 크기가 고정된다.

long형은 항상 4byte로서 32bit Windows체계의 int와 같고 -2,147,483,648～2,147,483,647범위를 가진다. 또한 long int라고 쓸수 있다. 프로그람을 MS-DOS나 Windows 3.1과 같은 16bit체계에서 실행한다면 long형을 지정하여 4byte옹근수형을 담보한다.

16bit체계에서 int형은 short형과 같은 범위를 가진다. 모든 체계에서 short형은 2byte를 차지하며 -32,768～32,767범위를 가진다. 현대 Windows체계에서 short형은 기억기호출속도가 느리므로 많이 사용되지 않는다. 그러나 두배나 더 큰 int형은

short형보다 더 빨리 호출할수 있다.

long형의 상수를 창조하려면 수값뒤에 문자 L을 쓴다.

```
longVar = 7678L;  // longVar에 long형상수 7678을 대입한다.
```

대부분의 번역프로그람은 사용비트수를 명백히 지적하는 옹근수형 즉 __int8, __int16, __int32, __int64을 제공한다. 매개의 형이름앞에는 두개의 밑줄이 있다. __int8은 char, __int16은 short, __int32는 long과 각각 같다. __int64는 19자리수의 huge옹근수를 보관한다.

제 5 절. 문자형

char형은 -128～127범위의 옹근수를 보관한다. char형변수는 오직 1byte의 기억기를 차지한다. 문자변수는 제한된 범위의 수값을 가지지만 ASCII문자보관에 더 많이 사용된다.

ASCII문자모임은 'a', 'B', '$', '3'과 같은 문자와 수자를 표시하는 한가지 방법이다. 이 수들은 0～127범위에 있다. 대다수 Windows체계들은 여러 나라의 자모와 도형문자를 사용하기 위하여 이 범위를 255까지 확장한다.

영어를 제외한 다른 나라 언어를 사용할 때 복잡성이 제기되며 같은 언어라도 콤퓨터체계들사이에 프로그람을 변환할 때에도 문제가 제기된다. 이것은 128～255범위의 문자들이 표준화되지 않았기때문이다. char형의 1byte체계는 여러 나라의 언어(례하면 조선어, 중국어, 일본어)로 문자들을 모두 표시할수 없다. 표준 C++는 이것을 조종하기 위하여 wchar_t라는 더 큰 문자형을 제공해준다.

1. 문자상수

문자상수에는 문자를 둘러싸는 단일인용표를 사용한다. 실례로 'a', 'b'는 2중인용표를 사용하는 문자렬상수와 다르다. C++번역프로그람은 문자상수와 만나면 그것을 대응하는 ASCII코드로 변환한다. 실례로 프로그람안에 있는 'a'는 97로 변환된다. (그림 2-4)

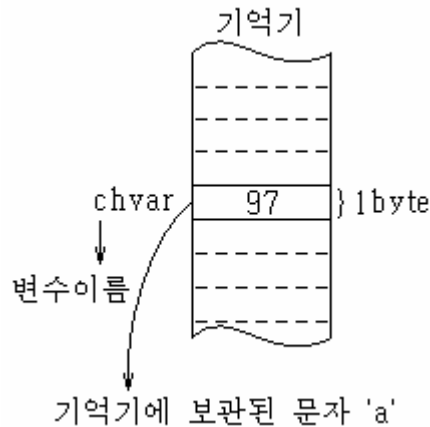



그림 2-4. 기억기안에서 char형변수

문자변수는 문자상수를 값으로 가질수 있다. 실례 2-4는 문자상수와 변수를 보여준다.

(실례 2-4) 문자변수

```
#include <iostream>
using namespace std;
int main()
{
   char charVar1 = 'A';        // char변수를 문자로서 정의한다.
   char charVar2 = '\t';       // char변수를 타브로서 정의한다.
   cout << charVar1;           // 문자를 표시한다.
   cout << charVar2;           // 문자를 표시한다.
   charVar1 = 'B';          // char변수를 문자상수로서 설정한다.
   cout << charVar1;           // 문자를 표시한다.
   cout << '\n';               // 행바꾸기문자를 출력한다.
   return 0;
}
```

- 초기화

변수는 정의할 때 초기화(initialization)할수 있다. 실례 2-4에서는 두개의 char형변수 charVar1과 charVar2를 문자상수 'A'와 '\t'로 초기화한다.

2. 확장문자

문자상수 '\t'는 상수로서 '\n'과 같이 확장문자의 하나의 실례이다. 확장문자(escape character)라는 이름은 역사선이 문자를 해석하는 일반적인 방법과 다른데로부터 생겨났다. 이 경우에 t는 문자 't'가 아니라 타브(tab)문자로 해석된다. 타브는 다음의 타브중지점으로부터 출력을 계속하게 한다. 콘솔방식프로그람에서 타브중지점은 각각 8개 공백씩 나가서 위치하고있다. 프로그람의 마지막 행의 cout에서는 다른 문자

상수 '\n'을 직접 출력한다.

확장문자는 개별적문자로서 사용하거나 문자렬상수에 포함하여 사용한다. 표 2-1은 확장문자를 보여준다.

역사선, 단일인용표, 2중인용표들이 상수안에서 쓰일 때에는 특별한 의미를 가지므로 그것들을 문자로 표시할 때 확장문자를 사용한다. 실례로

```
cout << "\"Run, Spot, run.\" she said."
```

표 2-1. 확장문자

확장문자	기능
\a	삑소리(beep)
\b	뒤걸음(backspace)
\f	인쇄용지바꾸기(formfeed)
\n	새 행(new line)
\r	복귀(return)
\t	타브(tab)
\\	역사선(back slash)
\'	인용표
\”	2중인용표
\xdd	16진수표기

이것은 다음과 같이 출력된다.

```
"Run, Spot, run." she said.
```

때로는 건반우에 없는 문자상수 례를 들면 ASCII코드 127과 같은 도형문자를 표시할 필요가 있다. 이때 '\xdd'표시를 사용한다. 여기서 매개의 d는 16진수이다. 실례로 흰4각형기호는 ASCII표에서 10진수로서 178이고 16진수로서 B2이다. 이 문자는 문자상수 '\xB2'이라고 쓸수 있다.

실례 2-4에서는 charVar1의 값 'A'와 charVar2의 값(타브)을 출력한다. 그다음 charVar1을 새로운 값 'B'로 설정하고 그것을 출력한 다음 마감에 새 행을 출력한다.

출력결과는 다음과 같다.

```
A	B
```

제 6 절. cin에 의한 입력

자료입력방법을 고찰하자. 실례 2-5는 사용자로부터 화씨온도를 받아들여 섭씨온도로 변환하고 그 결과를 표시한다. 여기서는 옹근수변수를 사용한다.

(실례 2-5) cin과 행바꾸기문자

```
#include <iostream>
using namespace std;
int main()
{
```

```
    int ftemp;
    cout << "화씨온도를 입력하시오:";
    cin >> ftemp;
    int ctemp = (ftemp - 32) * 5 / 9;
    cout << "섭씨온도는 " << ctemp << '\n';
    return 0;
}
```

명령문

```
cin >> ftemp;
```

는 사용자가 수를 입력할 때까지 프로그람이 기다리게 한다. 입력된 수는 변수 ftemp에 넣어진다. 예약어 cin("C in")은 C++에서 표준입력스트림에 대응하며 미리 정의되여있는 객체이다. 이 스트림은 건반으로부터 오는 자료를 표시한다. >>를 발취(extraction) 혹은 입력(get from)연산자라고 한다. 이 연산자는 왼변에 있는 스트림객체로부터 값을 받아들여 오른변에 있는 변수에 넣는다. 프로그람과의 대화는 다음과 같다.

```
화씨온도를 입력하시오: 212
섭씨온도는 100
```

그림 2-5는 cin과 발취연산자 >>를 사용한 입력을 보여준다.

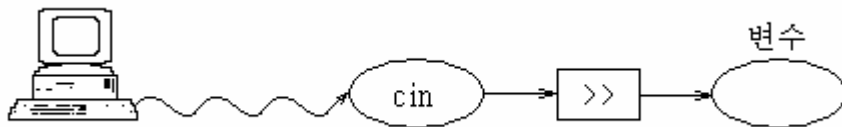



그림 2-5. cin에 의한 입력

1. 사용의 견지에서 정의된 변수들

실례 2-5에서 변수 ctemp는 프로그람의 선두가 아니라 여러행 뒤에서 정의되며 산수연산결과를 보관하는데 사용된다. 이미 설명한것처럼 프로그람의 어디서나 변수를 정의할수 있다. 그러나 대부분의 언어들에서는 변수를 첫 실행명령문앞에서 정의해야 한다.

함수안의 여러곳에서 사용하는 변수들은 함수의 선두에서 정의하는것이 좋다.

2. 삽입연산자 <<의 다중사용

실례 2-5에서 삽입연산자 <<는 두번째 cout명령문에서 다시 사용된다.

프로그람은 cout에 우선 "섭씨온도는 "을 보내고 그다음 ctemp의 값을 보내고 끝으로 새행문자 '\n'를 보낸다.

같은 방법으로 발취연산자 >>를 cin과 함께 다중사용하여 사용자는 값들의 렬을 입력할수 있다.

그러나 이런 입력은 자주 쓰이지 않는다. 그것은 입력사이에서 사용자가 반복입력

할 기회를 주지 않기때문이다.

3. 식

계산을 지정하는 변수, 상수, 연산자들의 배렬을 식(expression)이라고 한다. 실례로 alpha + 12와 (alpha - 37) * beta / 2는 식이다. 식을 처리한 결과는 보통 값이다. 따라서 alpha가 7이라면 첫식은 값 19로 된다.

식의 부분 역시 식일수 있다. 둘째 실례에서 alpha - 37과 beta / 2는 식이다. 한편 단일한 변수와 상수(alpha와 37 등)도 식으로 본다.

식은 명령문이 아니다. 명령문은 번역프로그람에게 반두점으로 끝나는 어떤 처리를 수행하게 하지만 식은 계산을 지정한다. 한개 명령문에는 여러개의 식이 있을수 있다.

4. 우선순위

식에 괄호를 넣을수 있다. 실례로 식

(ftemp - 32) * 5 / 9

에서 괄호가 없으면 *가 -보다 우선순위(precedence)가 더 높으므로 곱하기가 먼저 수행된다. 괄호가 있으면 그안의 연산이 먼저 수행되므로 먼저 덜기를 하고 그다음 곱하기를 한다.

*와 /의 우선순위는 같고 왼쪽연산자가 먼저 수행되므로 곱하기를 하고 그다음 나누기를 한다. 보통 우선순위와 괄호는 대수와 기타 프로그람언어에서와 같이 적용된다.

제 7 절. 류동소수점수형과 론리형

int와 char는 둘다 옹근수이므로 소수부를 가지지 않는다. 수를 류동소수점수형변수에 보관하는 방법을 고찰하자.

류동소수점수형(floating)변수는 3.1415927, 0.0000625, -10.2와 같은 수자들의 렬로서 수를 표시한다. 류동소수점수는 소수점의 왼쪽에 옹근수부를, 오른쪽에 소수부를 가진다. 류동소수점수형변수들은 수학의 실수를 표시하며 거리, 면적, 온도 등 소수부를 가지는 량을 표시하는데 사용된다.

C++에는 세가지 종류의 류동소수점수 즉 float형, double형, long double형이 있다.

먼저 크기가 제일 작은 float형을 고찰해보자.

1. float형

float형은 약 $3.4\times10^{-38}\sim3.4\times10^{38}$범위의 수를 7자리의 정확도로 보관한다. float형은 4byte의 기억기를 차지한다. (그림 2-6)

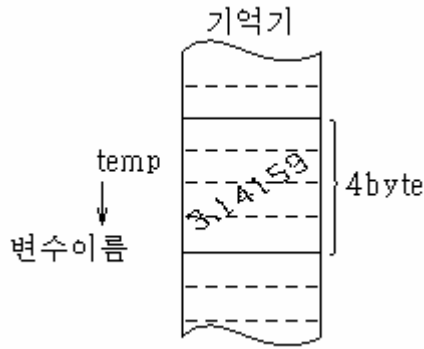



그림 2-6. 기억기에서 float형변수

실례 2-6은 원의 면적을 계산하고 화면에 표시한다.

(실례 2-6) 류동소수점수형변수

```
#include <iostream>
using namespace std;
int main()
{
  float rad;                                  // float형변수
  const float PI = 3.14159F;                  // const float형
  cout << "원의 반경을 입력하시오:";
  cin >> rad;
  float area = PI * rad * rad;
  cout << "원의 넓이는 " << area << endl;
  return 0;
}
```

프로그람의 실행결과는 다음과 같다.

```
원의 반경을 입력하시오: 0.5
원의 넓이는 0.785398
```

2. double형과 long double형

다른 류동소수점수형으로서 double과 long double은 float보다 더 큰 기억공간을 요구하며 더 넓은 값범위와 정확도를 가진다.

double형은 8byte의 기억기를 요구하고 1.7×10^{-308}~1.7×10^{308}범위의 수를 15자리의 정확도로 보관한다. long double형은 10byte를 요구하고 약 1.2×10^{-4932}~1.2×10^{4932}범위의 수를 19자리의 정확도로 보관한다.

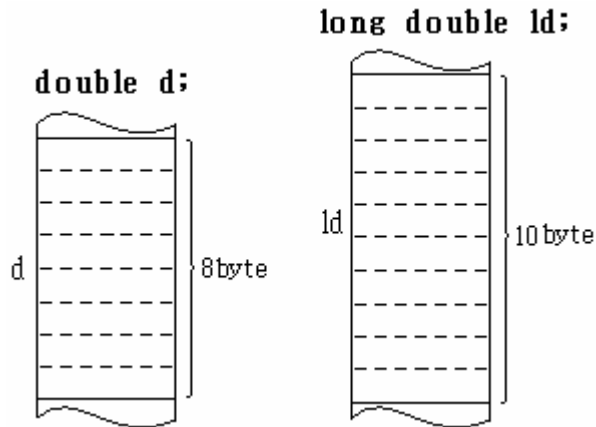



그림 2-7. double과 long double형의 변수

3. 류동소수점상수

실례 2-6에서 수 3.14159F는 류동소수점수형상수의 례이다. 소수점은 옹근수가 아니라 류동소수점상수라는것을 알려주며 F는 그것이 double이나 long double이 아니라 float형이라는것을 알려준다. 수는 표준10진표기법으로 쓴다. double형의 상수에는 뒤붙이를 쓰지 않는다. long double형일 때에는 뒤붙이 L을 쓴다.

또한 류동소수점상수를 지수표기법으로 쓸수 있다. 지수표기는 령을 많이 쓰지 않고 큰 수를 표시하는 방법이다. 실례로 1000000000은 지수표기로 1.0E9라고 쓸수 있다. 마찬가지로 1234.56은 1.23456E3이라고 쓸수 있다. E뒤에 오는 수를 지수라고 한다. 지수는 보통산수표기로 수를 표시할 때 소수점을 몇자리 이동하여야 하는가를 표시한다.

지수는 정수 혹은 부수이다. 지수표기의 수 6.35239E-5는 소수표기 0.0000635239와 같다. 이것은 6.35239의 10^{-5}배와 같다.

4. const변경자

실례 2-6에서는 float형변수와 함께 변경자 const를 사용한다. 즉 명령문에 다음과 같이 쓰고있다.

```
const float PI = 3.14159F;        // const float형
```

예약어 const는 변수의 자료형앞에 배치한다. 이것은 변수값이 프로그람을 실행하는 전기간 변하지 않는다는것을 의미한다. const변경자로 정의한 변수값을 변경하려고 하면 번역프로그람은 오유를 통보한다.

const는 프로그람에서 실수하여 변수를 변경하지 못하게 한다. (실례에서 PI값을 들수 있다.) const변경자는 단순변수가 아닌 다른 실체들에도 적용할수 있다.

5. #define지령

C++에서는 일반적으로 앞처리지령 #define을 사용하여 상수를 지정할수 있다. 례를 들면 다음 행

```
#define PI 3.14159
```

를 프로그람의 선두에 넣으면 후에 PI라는 식별자를 모두 3.14159로 교체한다.

이 방법은 C에서 자주 사용된다. 그러나 #define을 사용한 자료형의 지정은 프로그람오유를 일으키는 경우가 있으므로 표준변수에 const를 붙여서 #define과 교체하고있다.

그러나 이와 같은 낡은 프로그람의 구조를 사용할수도 있다.

6. bool형

int형의 변수는 수십억개의 가능한 수들을 가지며 char형은 256개의 수를 가진다.

bool형의 변수들은 오직 두개의 가능한 수 true와 false를 가진다.

리론적으로 bool형은 오직 1bit의 기억기를 요구하지만 실천적으로 번역프로그람이 옹근수로부터 개별적인 비트를 끌어내는데 보충적인 시간이 걸리므로 빨리 호출할수 있도록 옹근수로서 보관한다.

bool형은 비교결과를 보관하는데 많이 쓰인다.

그러면 alpha가 beta보다 작은가?

작으면 bool값으로서 true값이 주어지고 작지 않으면 false값이 주어진다.

bool형은 19세기의 영국수학자 George Boole로부터 유래되였다. 그는 참 혹은 거짓값을 가지는 론리연산자를 사용하는 개념을 도입하였다. 그리하여 참 또는 거짓값을 자주 Boolean값이라고도 한다.

제 8 절. setw조작자

조작자가 자료표시방법을 변경하거나 조작하기 위하여 삽입연산자 <<와 함께 사용되는 연산자라는데 대하여 이미 언급하였다. 이번에는 endl조작자외에 다른 조작자로서 출력마당폭을 변경하는 setw를 고찰해보자.

cout에 의해 표시되는 어느 한 마당을 차지하는 값 즉 일정한 폭을 가지는 가상적인 칸들을 고찰할수 있다. 기정마당은 값을 보관하는데 충분한 폭이다. 즉 옹근수 567은 3문자폭의 마당을 차지하며 문자렬 "project"는 7개 문자너비의 마당을 차지한다. 그러나 일부 경우에 이것은 좋은 결과를 가져다주지 않는다. 례를 들어보자. 실례 2-7은 첫째 렬에 지역의 이름을 출력하고 둘째 렬에 그 인구수를 출력한다.

(실례 2-7) setw조작자의 필요성

```
#include <iostream>
```

```
using namespace std;
int main()
{
  long pop1 = 2425785, pop2 = 105000, pop3 = 3761;
  cout << "지역 " << "인구" << endl
       << "ㄱ시 " << pop1  << endl
       << "ㄴ군 " << pop2  << endl
       << "ㄷ리 " << pop3  << endl;
  return 0;
}
```

이 프로그람의 출력결과는 다음과 같다.

```
지역 인구
ㄱ시 2425785
ㄴ군 105000
ㄷ리 3761
```

이러한 출력은 수들을 비교하기 힘들게 한다. 그러므로 오른쪽으로부터 렬들을 채워쓰면 좋다. 또한 지역이름을 수들과 구별하기 위하여 공백을 삽입하여야 한다.

실례 2-7을 수정한 실례 2-8에서는 setw조작자를 사용하여 이름과 수값들의 마당폭을 지정하여 이 문제를 해결한다.

(실례 2-8) setw조작자

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
  long pop1 = 2425785, pop2 = 105000, pop3 = 3761;
  cout << setw(8) <<"지역 " << setw(12) << "인구" << endl
       << setw(8) << "ㄱ시 " << setw(12) << pop1  << endl
       << setw(8) << "ㄴ군 " << setw(12) << pop2  << endl
       << setw(8) << "ㄷ리 " << setw(12) << pop3  << endl;
  return 0;
}
```

setw조작자는 n문자길이의 마당안에 뒤에 오는 수(혹은 문자렬)가 출력되도록 한다. 여기서 n은 setw(n)의 인수로서 마당의 폭을 정의한다. 그림 2-8은 이것을 보여준다. long형은 인구수의 출력에 사용되며 2byte옹근수형을 사용하는 체계에서 자리넘침오유를 방지한다.

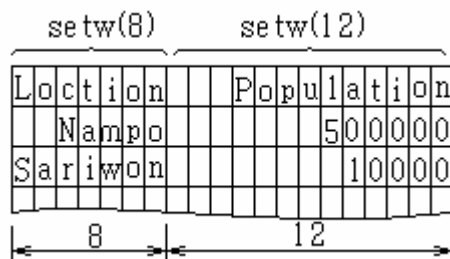



그림 2-8. 마당의 폭과 setw

실례 2-8의 출력은 다음과 같다.

```
지역      인구
ㄱ시   2425785
ㄴ군    105000
ㄷ리      3761
```

1. 삽입연산자의 다중사용

실례 2-7과 실례 2-8에서는 cout명령문을 여러행으로 나누어 쓰고있다.

변수 pop1, pop2, pop3은 정의와 함께 초기화된다. 이것은 실례 2-4에서 char변수를 초기화하는 방법과 비슷하다. 그러나 여기서는 하나의 long예약어를 사용하면서 반점으로 구분하여 한행에서 3개의 변수를 모두 정의하고 초기화한다.

이것은 여러개의 변수가 같은 형을 가지는 경우에 기억공간을 절약한다.

2. IOMANIP머리부파일

endl을 제외한 조작자의 선언은 IOSTREAM이 아니라 IOMANIP라는 머리부파일에 있다. 조작자를 사용할 때에는 반드시 프로그람에 이 머리부파일을 포함하여야 한다.

제 9 절. signed와 unsigned형

지금까지의 실례들에서는 4개의 형 int, char, float, long을 사용하였다. 또한 short, double, long double에 대하여 언급하였다. 표 2-2는 형정의에 쓰이는 예약어와 그 값범위, 정확도와 기억바이트수를 보여준다.

표 2-2. C++의 기본자료형

예약어	수값범위		정확도 (자리수)	기억 바이트수
	아래한계	웃한계		
char	-128	127	-	1
short	-32,768	32,767	-	2
int	-2,147,483,648	2,147,483,647	-	4
long	-2,147,483,648	2,147,483,647	-	4
float	3.4×10-38	3.4×1038	7	4
double	1.7×10-308	1.7×10308	15	8
long double	3.4×10-4932	3.4×104932	19	10

- unsigned자료형

문자형과 옹근수형의 부호를 제거하고 그 범위를 0으로부터 시작하여 오직 정수만 포함하도록 변경할수 있다. 이것은 signed형보다 2배나 더 큰 수를 표시하게 한다. 표

2-3에 unsigned형을 주었다.

표 2-3. unsigned옹근수형

예약어	수값범위		기억 byte수
	아래한계	웃한계	
unsigned char	0	255	1
unsigned short	0	65,535	2
unsigned int	0	4,294,967,295	4
unsigned long	0	4,294,967,295	4

unsigned형들은 항상 정의 량을 표시할 때 사용된다. 또한 signed형들은 정의 범위가 너무 크지 않을 때 사용된다.

옹근수형을 unsigned형으로 변경하려면 자료형 예약어앞에 예약어 unsigned를 써야 한다. 실례로 char형의 unsigned변수는 다음과 같이 정의한다.

```
unsigned char ucharVar;
```

signed형의 범위를 초과하면 프로그람오유가 생긴다. 이 오유는 일부 경우에 unsigned형을 사용하여 제거할수 있다. 실례 2-9는 상수 1500000000을 signedVar에 int형으로서, unsignVar에 unsigned int로서 보관한다.

(실례 2-9) signed와 unsigned옹근수

```
#include <iostream>
using namespace std;
int main()
{
    int signedVar = 1500000000;
    unsigned int unsignVar = 1500000000;
    signedVar = (signedVar * 2) / 3;
    unsignVar = (unsignVar * 2) / 3;
    cout << "SignedVar = " << signedVar << endl;    // 오유
    cout << "UnsignVar = " << unsignVar << endl;    // 옳다
    return 0;
}
```

프로그람은 두 변수에 2를 곱하고 그것들을 3으로 나눈다. 결과는 본래의 수보다 작지만 도중계산결과는 원래수보다 크다. 이것은 오유를 일으킨다. 실례 2-9에서는 원래수 1500000000을 2/3하여 다시 보관하려고 한다. 그러므로 signedVar에서는 3000000000이라는 결과가 나오고 int변수의 범위(-2,147,483,648～2,147,283,647)를 초과한다. 결과는 다음과 같다.

```
signedVar = -431,655,965
unsignedVar = 1,000,000,000
```

현재 signed변수는 틀린 답을 표시한다. unsigned변수는 곱한 결과를 보관할수 있을 정도로 충분히 크므로 정확한 결과를 기록한다.(이 결과는 16bit 또는 64bit콤퓨터에서 int형의 byte수가 다르므로 서로 다른 결과를 가져온다.)

제 10 절. 형변환

C++에서는 여러가지 자료형을 포함하는 식을 다루는 능력이 일부 다른 언어들보다 강화되였다. 실례 2-10을 고찰하자.

(실례 2-10) 혼합식

```
#include <iostream>
using namespace std;
int main()
{
   int count = 7;
   float avgWeight = 155.5F;
   double totalWeight = count * avgWeight;
   cout << "TotalWeight = " << totalWeight << endl;
   return 0;
}
```

여기서는 int형변수와 float형변수를 곱하여 double형의 결과를 얻는다. 이 프로그람은 오유없이 번역된다. 번역프로그람은 다른 형의 수들을 곱하는것으로 간주한다.

그러나 모든 언어들이 다 그런것은 아니다. 일부 언어들은 혼합식을 허용하지 않으며 실례 2-10의 산수계산명령문을 오유로 처리한다.

그러나 C++와 C는 혼합식을 쓰는 원인이 있다고 가정하고 사용자의 의도를 실현하려고 하였다. 이것은 C++와 C가 인기를 끄는 원인의 하나로서 프로그람작성에서 자유도를 준다. 물론 자유도가 높을수록 오유를 범할수 있는 기회가 더 많아진다.

1. 자동형변환

번역프로그람이 실례 2-10과 같은 혼합식을 만났을 때 어떻게 동작하는가를 고찰하자. 형에는 높은 형과 낮은 형이 있다. 이것은 표 2-4에 보여준 순서에 따른다.

+나 *와 같은 산수연산자들은 같은 형의 두개 연산수에 적용된다. 한개 식에서 형이 다른 두개 연산수를 만나면 낮은 형의 변수는 높은 형의 변수로 변환된다.

표 2-4. 자료형의 순위

자료형	순위
long double	제일 높다.
double	↑
float	
long	
int	
short	↓
char	제일 낮다.

이리하여 실례 2-10에서 int형의 count값은 float형으로 변환되여 float변수 avgWeight와 곱하기 전에 림시변수에 보관된다. 결과(아직은 float형)는 그다음 double로 변환되여 double형변수 totalWeight에 대입된다. 그 과정을 그림 2-9에 보여준다.

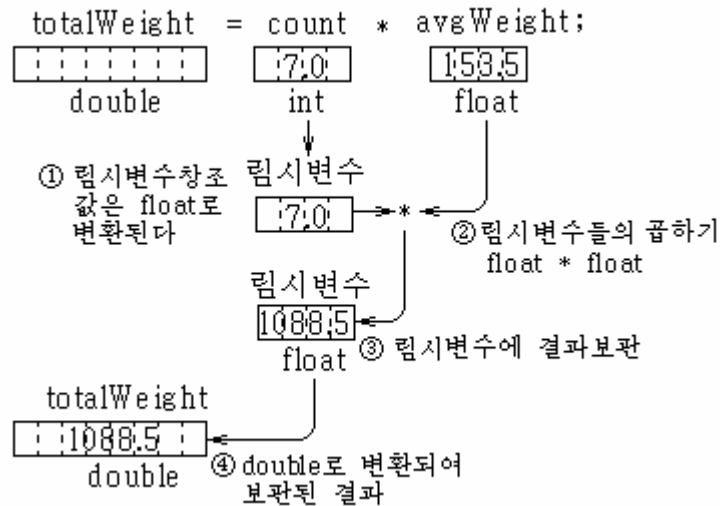



그림 2-9. 자료형변환

이 변환은 암시적으로 진행된다. 즉 C++가 자동적으로 수행한다. 그러나 흔히 번역프로그람은 이러한 관례를 만족스러운것으로 여기지 않는다. 또한 객체를 사용하는 경우에는 자체의 자료형을 정의하는것이 효과적이다. 혼합식에서 새 자료형을 사용하려고 한다면 어떤 형의 객체를 다른 형의 객체로 변환하는 자체의 형변환루틴을 정의하여야 한다. 번역프로그람은 기본자료형에 대하여서만 변환을 수행한다.

2. 강제형변환

C++에서 강제형변환은 자동자료변환과 달리 작성자가 지정하는 자료변환에 사용된다.

대체로 번역프로그람이 어떤 형으로부터 다른 형으로 값을 변환할수 없을 때 작성자가 변환해야 하는 경우에 강제형변환을 사용한다.

표준 C++의 강제형변환에는 정적강제형변환(static_cast), 동적강제형변환(dynamic_cast), 재해석강제형변환(reinterpret_cast), 상수강제형변환(const_cast)이 있다. 여기서는 정적강제형변환만 설명한다.

여기에 C++강제형변환을 사용하여 int형변수를 char형으로 변환하는 명령문이 있다.

```
aCharVar = static_cast<char>(anIntVar);
```

여기서는 강제형변환하려는 변수(anIntVar)를 괄호안에 넣고 그것을 변경하려는 형(char)을 각괄호 <>안에 넣는다.

이때 anIntVar는 aCharVar에 대입되기 전에 char형으로 변경된다.

강제형변환이 요구되는 경우가 있다. 실례 2-9에서는 중간결과가 변수형의 용량을 초과한다. 이 경우에는 int대신에 unsigned int를 사용하여 문제를 해결한다. 따라서 중간결과 3000000000은 부호없는 변수의 범위안에 놓인다.

그러나 중간결과가 부호없는 형의 범위도 벗어나면 강제형변환을 사용하여 문제를 해결한다. (실례 2-11)

(실례 2-11) signed와 unsigned옹근수

```
#include <iostream>
using namespace std;
int main()
{
   int intVar = 1500000000;
   intVar = (intVar * 10) / 10;                    // 결과가 너무 크다
   cout << "intVar = " << intVar << endl;     // 오유
   intVar = 1500000000;
   intVar = (static_cast<double>(intVar) * 10) / 10;
   cout << "intVar = " << intVar << endl;     // 옳다
   return 0;
}
```

변수 intVar에 10을 곱한 결과 15000000000은 int 혹은 unsigned int형변수에 맞지 않게 너무 크다. 그러므로 프로그람의 앞부분에서는 옳지 않은 답이 얻어진다.

변수의 자료형을 double로 다시 정의하면 15자리까지의 수를 보관할수 있으므로 여유가 생긴다. 그러나 프로그람이 기억기를 적게 소비하도록 하기 위하여 변수를 double형으로 변경하지 않는다. 이 경우에 다른 해결방법이 있다. 즉 intVar를 곱하기 전에 double형으로 강제형변환하는것이다. 이것을 형강제형변환(coercion)이라고하며 자료는 다른 형으로 강제형변환된다. 식

```
static_cast<double>(intVar);
```

는 intVar를 double형으로 강제형변환한다. 이것은 intVar와 같은 값을 가지는 double형의 림시변수를 만들고 그 변수에 10을 곱한다. 림시변수는 double형이므로 결과를 충분히 보관할수 있다. 그다음 결과를 10으로 나누고 int변수 intVar에 대입한다.

프로그람의 출력은 다음과 같다.

```
intVar = 211509811
intVar = 1500000000
```

첫째 답은 강제형변환이 없으므로 틀린다. 그러나 둘째 답은 강제형변환에 의해 옳은 결과로 된다.

표준 C++이전에는 강제형변환에 다른 형식을 사용하였다. 즉

```
aChar = static_cast<char>(anIntVar);
```
대신에
```
aChar = (char)anIntVar;
```
또는
```
aChar = char(anIntVar);
```
이 수법은 리해하기 힘들고 원천코드편집기의 Find조작을 사용하여 탐색하기 힘들다. static_cast는 이 문제를 해결해준다.

제 11 절. 산수연산자

C++는 4개의 산수연산자 +, -, *, /를 더하기, 덜기, 곱하기, 나누기에 사용한다. 이 연산자들은 모든 자료형에서 쓰인다. 다른 언어들처럼 산수연산자를 제일 많이 사용하며 사용법은 대수에서와 비슷하다. 그밖에 다른 산수연산자도 있다.

1. 나머지연산자

나머지연산자는 옹근수변수(char, short, int, long)에 대하여서만 작용하는 산수연산자이며 %로 표시된다. 나머지연산자는 어떤 수를 다른 수로 나눈 나머지를 얻는다. 실례 2-12는 나머지연산자에 대하여 보여준다.

(실례 2-12) 나머지연산자

```
#include <iostream>
using namespace std;
int main()
{
  cout <<  6 % 8 << endl
       <<  7 % 8 << endl
       <<  8 % 8 << endl
       <<  9 % 8 << endl
       << 10 % 8 << endl;
  return 0;
}
```
실례에서는 수 6～10을 나머지연산자를 사용하여 8로 나눈다. 답은 6,7,0,1,2이다. 우선순위와 관련하여 다음의 식
```
cout << 6 % 8;
```
에서 나머지연산자는 <<연산자보다 높은 우선권을 가지므로 먼저 평가된다.

2. 산수대입연산자

C++는 코드를 줄이는 방법을 몇가지 제공해준다. 그중 하나가 산수대입연산자를 리용한 방법이다.

일반적으로 대부분의 언어들에서 다음과 같은 형식의 명령문을 많이 사용한다.

```
total = total + item;
```

이것은 현재값에 어떤 값을 더하는 명령문인데 여기서는 total이름이 두번 나타나므로 코드가 길어진다.

C++는 더 간단한 수법으로서 산수대입연산자를 제공해준다.

산수(arithmetic)대입연산자는 산수연산자와 대입(assignment)연산자의 결합이며 연산수의 반복이 없다. 산수연산자를 사용하여 우의 식과 꼭같은 명령문을 쓰면 다음과 같다.

```
total += item;
```

그림 2-10은 두 식의 등가성을 보여준다.

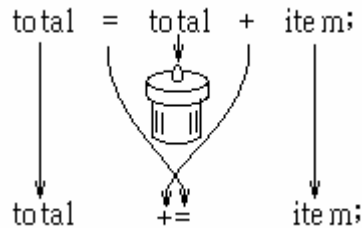



그림 2-10. 산수대입연산자

모든 산수연산자들에 대응하는 산수대입연산자들이 있다. 즉 +=,-=,*=,/=,%=.

(실례 2-13) 산수대입연산자

```
#include <iostream>
using namespace std;
int main()
{
   int ans = 27;
   ans += 10;
   cout << ans << ", ";
   ans -= 7;
   cout << ans << ", ";
   ans *= 2;
   cout << ans << ", ";
   ans /= 3;
   cout << ans << ", ";
   ans %= 3;
   cout << ans << endl;
   return 0;
}
```

출력결과는 다음과 같다.

```
37, 30, 60, 20, 2
```

3. 증가연산자

때때로 현존 변수값에 1을 더할 필요가 제기된다. 그 일반적인 방법은

```
count = count + 1;
```

산수대입연산자를 사용하면

```
count += 1;
```

더 간단한 방법은

```
++count;
```

++연산자는 연산수를 하나 증가시킨다.

- 앞붙이(prefix)와 뒤붙이(postfix)형식

증가(incremental)연산자는 두가지 방법 즉 앞붙이(변수앞에 연산자를 놓는다.) 혹은 뒤붙이(변수의 뒤에 연산자를 놓는다.)로서 사용된다. 그러면 두 연산자의 차이는 무엇인가? 변수는 가끔 다른 연산을 처리하는 식안에서 증가되군 한다. 례를 들면

```
totalWeight = avgWeight * ++count;
```

여기서 문제는 곱하기연산이 count가 증가되기 전에 진행되는가 혹은 증가된 다음에 진행되는가 하는것이다. 이 경우에는 count가 먼저 증가된다. 그것은 앞붙이형식의 ++count를 사용하기때문이다. 뒤붙이형식의 count++를 사용하면 곱하기연산이 먼저 수행되고 그다음 count가 증가된다. 이것을 그림 2-11에서 보여준다.

그림 2-11. 증가연산자

실례 2-14는 앞붙이와 뒤붙이형식의 증가연산자를 보여준다.

(실례 2-14) 증가연산자

```
#include <iostream>
using namespace std;
int main()
{
  int count = 10;
  cout << "count = " << count << endl;
  cout << "count = " << ++count << endl; // 앞붙이
  cout << "count = " << count << endl;
  cout << "count  = " << count++ << endl; // 뒤붙이
```

```
    cout << "count   = " << count << endl;
    return 0;
}
```

출력결과는 다음과 같다.

```
count = 10
count = 11
count = 11
count = 11
count = 12
```

먼저 count가 증가된다. 앞붙이형식의 ++연산자를 사용하므로 명령문평가의 시초에 즉 출력조작이 수행되기 전에 증가된다. 식 ++count의 값이 표시될 때 count는 이미 증가되여있고 <<는 11을 표시한다. 다음으로 뒤붙이형식의 ++연산자에 의하여 count를 증가시킨다. 식 count++가 표시될 때 count의 증가되지 않은 값 11이 남아있고 명령문의 실행이 끝난후 증가되므로 프로그람의 마지막 명령문에서 count는 값 12을 가진다.

- 감소(decremental)연산자

감소연산자 --는 연산수로부터 1을 더는것을 제외하면 증가연산자와 비슷하다. 또한 앞붙이와 뒤붙이형식이 있다.

제 12 절. 서고함수

C++의 많은 동작은 서고함수(library function)에 의해 수행된다. 서고함수는 함수호출, 수학계산, 자료변환 등을 진행한다.

실례 2-15는 서고함수 sqrt()를 사용하여 사용자가 입력한 수의 루트를 계산한다.

(실례 2-15) sqrt()서고함수

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    double number, answer;
    cout << "수를 입력하십시오:";
    cin >> number;
    answer = sqrt(number);
    cout << "루트=" << answer << endl;
    return 0;
}
```

우선 프로그람은 사용자로부터 수를 하나 얻는다. 그다음 그 수를 sqrt()함수의 인수로 사용한다. 즉

```
answer = sqrt(number);
```

인수는 함수에 대한 입력으로서 함수이름뒤의 소괄호안에 넣는다. 그다음 함수는

인수를 처리하고 값을 돌려주는데 이 값은 함수로부터의 출력이다. 이 경우에 돌림값은 원래수의 루트이다. 값을 돌려주는것은 함수식이 값을 가진다는것을 의미한다. 그다음 다른 변수(이 경우에는 answer)에 대입하고 값을 표시한다. 출력결과는 다음과 같다.

```
수를 입력하십시오: 1000
루트= 31.622777
```

함수에 대한 인수와 돌림값은 정확한 자료형이여야 한다. 번역프로그람의 방조파일에서 서고함수를 검색하여 함수와 자료형을 얻을수 있다. sqrt()일 때에는 인수와 돌림값이 둘다 double형이다.

1. 머리부파일과 서고파일

- 머리부파일

cout와 마찬가지로 서고함수를 사용하려면 그 객체가 들어있는 머리부파일을 포함하여야 한다. sqrt()함수는 머리부파일 CMATH에 선언되여있다. 실례 2-15에서 앞처리지령

```
#include <cmath>
```

는 원천파일에 이 머리부파일을 포함시킨다.

서고함수를 사용할 때 적당한 머리부파일을 포함하지 않으면 아래와 같은 오유통보문이 출력된다.

```
"sqrt" : undefined identifier
```

- 서고파일

실행파일을 창조할 때 프로그람에는 서고함수와 객체들을 포함하고있는 여러개의 파일들이 련결된다. 이 파일들에는 기계어로 된 함수들의 실행가능코드가 들어있다. 서고파일은 보통 확장자 .LIB를 가진다. sqrt()함수는 바로 서고파일에 있다. 함수는 련결프로그람에 의해 서고파일로부터 자동적으로 추출되여 실례 2-15의 프로그람으로부터 호출할수 있게 적당히 결합된다.

- 머리부파일과 서고파일사이의 관계

서고파일과 머리부파일사이의 관계는 복잡해질수 있다. sqrt()와 같은 서고함수를 사용하자면 그것을 포함하고있는 서고파일을 프로그람에 련결하여야 한다. 련결프로그람에 의해 서고파일로부터 적당한 함수들이 프로그람에 결합된다.

사용자의 원천파일에 있는 함수들은 서고파일안의 함수들과 기타 요소들의 형과 이름을 알아야 하는데 그 정보는 머리부파일에 주어진다.

매개 머리부파일에는 일정한 부류의 함수들에 대한 정보가 포함되여있다. 함수들은 하나의 서고파일로 묶어져있지만 그 정보는 여러개의 머리부파일들에 갈라져있다. IOSTREM머리부파일에는 각종 입출력함수들과 cout를 비롯한 객체들에 대한 정보가 들어있고 CMATH머리부파일에는 sqrt()와 같은 수학함수들에 대한 정보가 들어있다.

그림 2-12는 머리부파일과 서고파일들, 프로그람개발에서 쓰이는 다른 파일들사이의 관계를 보여준다. C++에서 머리부파일의 사용은 보편적이다. 서고함수를 사용하거나 미리 정의된 객체이나 연산자를 사용할 때에는 항상 해당한 선언이 들어있는 머리부파일을 사용해야 한다.

2. #include를 사용하는 두가지 방법

#include는 두가지 방법으로 사용할수 있다.

실례 2-15에서 IOSTREAM과 CMATH를 둘러싸는 각괄호 < >는 번역프로그람이 표준 INCLUDE등록부로부터 이 파일검색을 시작하여야 한다는것을 가리킨다. INCLUDE등록부에는 번역프로그람제작자에 의하여 제공된 체계용의 머리부파일들이 들어있다.

파일이름지적에 각괄호대신 2중인용표를 사용할수 있다. 실례로

```
#include "myheader.h"
```

인용표는 번역프로그람에게 현재 등록부에서 머리부파일을 찾을것을 지시한다. 현재 등록부는 원천파일들을 포함하고있는 등록부이다.

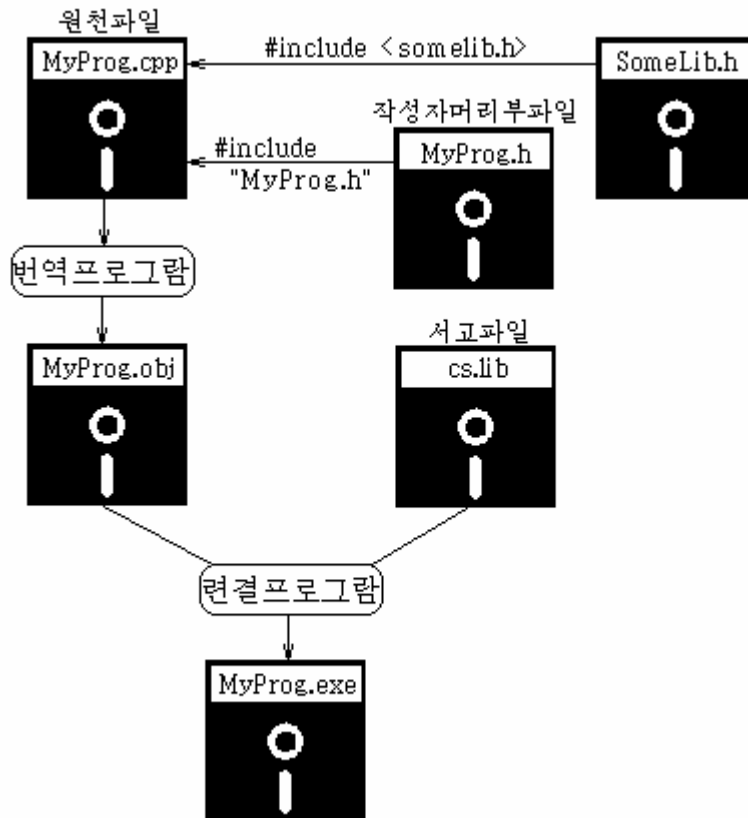



그림 2-12. 머리부파일과 서고파일

요 약

이 장에서는 C++프로그람의 기본구성요소인 함수에 대하여 배웠다. main()함수는 항상 프로그람이 실행될 때 처음으로 실행된다.

함수는 명령문들로 이루어지며 명령문은 콤퓨터가 어떤 일을 수행하게 한다. 매개 명령문은 한개이상의 식을 포함할수 있다. 식은 특정한 값으로 평가되는 변수와 연산자들의 렬이다.

보통 출력은 C++의 cout객체와 삽입연산자 <<를 가지고 조종한다. 이것은 표준출력장치(보통 화면)에 변수와 상수를 보내게 한다. 입력은 cin과 발취연산자 >>로 조종한다. 이것은 표준입력장치(건반)로부터 값을 받아들인다.

C++에는 여러가지 자료형들이 준비되여있다. char, int, long, short는 옹근수형, float, double, long double은 류동소수점수형이다. 이런 형의 자료는 signed형이다. 부호없는 옹근수형은 예약어 unsigned로 표시하며 부수값을 보관하지 않으므로 두배나 더 큰 옹근수를 보관한다. bool형은 론리변수에 사용되며 true 혹은 false만 가진다.

const예약어는 변수값이 프로그람의 실행과정에 변하지 않는다는것을 담보한다.

혼합식에서 변수는 자동적으로 어떤 형으로부터 다른 형에로 변환되며 작성자는 강제형변환을 지정할수 있다.

C++는 일반산수연산자로서 +, -, *, /와 함께 나머지연산자 %를 가진다. %는 옹근수나누기의 나머지를 돌려준다.

산수대입연산자 +=, -= 등은 산수연산과 대입을 동시에 처리한다. 증가 및 감소연산자 ++와 --는 변수를 하나 증가시키거나 감소시킨다.

앞처리지령은 번역프로그람에 대한 명령이다.

#include지령은 번역프로그람이 현존 원천파일에 다른 파일을 삽입하게 하며 #define지령은 어떤 기호를 다른것이 대신하게 한다. using지령은 번역프로그람이 일정한 이름공간안의 이름들을 인식하게 한다.

서고함수를 사용하면 서고파일에 있는 함수코드가 자동적으로 프로그람과 결합된다. 함수선언을 포함하고있는 머리부파일은 #include지령에 의하여 원천파일에 삽입된다.

문 제

1. 하나의 프로그람을 함수들로 분할하는것은

ㄱ) 객체지향프로그람작성을 위한 열쇠이다.

ㄴ) 프로그람을 개념화하기 쉽게 한다.

ㄷ) 프로그람의 크기를 줄인다.

ㄹ) 프로그람을 고속으로 실행할수 있게 한다.

2. 함수이름은 _______앞에 놓는다.

3. 함수본체는 ______에 의하여 구분된다.

4. main()함수를 왜 특수함수라고 하는가?

5. 콤퓨터에게 어떤 일을 하게 하는 C++지령을 ______이라고 한다.

6. 표준 C++의 설명문과 C의 설명문의 실례를 각각 하나씩 드시오.

7. 식은

ㄱ) 보통 수값을 평가한다.

ㄴ) 항상 함수밖에 있다.

ㄷ) 명령문의 일부분일수 있다.

8. 32bit체계에서 다음의 자료형은 몇 byte를 차지하는가?

ㄱ) int형

ㄴ) long double형

ㄷ) float형

ㄹ) long형

9. char형변수는 값 301을 가질수 있는가?

10. 다음것은 어떤 종류의 프로그람요소인가?

ㄱ) 12

ㄴ) 'a'

ㄷ) 4.28915

ㄹ) Kim

ㅁ) Kim()

11. 화면에 다음과 같이 출력하는 명령문을 쓰시오.

ㄱ) 문자 'x'

ㄴ) 이름 Kim

ㄷ) 수 509

12. 대입명령문에서 같기기호의 왼변값은 오른변값과 항상 같은가?

13. 10문자폭의 마당에 변수 var를 표시하는 명령문을 쓰시오.

14. cout와 cin을 사용하려면 원천파일에 어떤 머리부파일을 포함하여야 하는가?

15. 건반으로부터 수값을 읽어들여 변수 temp에 넣는 명령문을 쓰시오.

16. setw를 사용하려면 어떤 머리부파일을 포함하여야 하는가?

17. 번역프로그람이 공백을 무시하는 규칙에 대한 두가지 례외는___와___이다.

18. 하나의 산수식에서 서로 다른 자료형을 사용하는것은 전적으로 옳은가?

19. 식 11 % 3은 ______로 평가된다.

20. 산수대입연산자는 어떤 두개의 연산자를 결합한것인가?

21. 산수대입연산자를 사용하여 변수 temp값을 23씩 증가시키는 명령문을 쓰시오. 또한 산수대입연산자를 사용하지 않고 같은 조작을 하는 명령문을 쓰시오.

22. 증가연산자는 변수값을 얼마나 증가시키는가?

23. var1이 20일 때 다음의 코드는 어떤 값을 출력하는가?

```
cout << var1--;
cout << ++var1;
```

24. 지금까지의 실례들에서 머리부파일은 어떤 목적에 사용되였는가?

25. 서고함수의 실제코드는 ______파일에 포함된다.

련습문제

1. 1평은 3.3m^2이다. m^2수를 사용자가 입력하면 그것을 등가한 평수로 표시하는 프로그람을 작성하시오.

2. 하나의 cout명령문을 사용하여 다음의 형식으로 출력하는 프로그람을 작성하시오.

```
1990      135
1991     7290
1992    11300
1993    16200
```

3. 다음과 같이 출력하는 프로그람을 작성하시오.

```
10
20
19
```

여기서 10은 옹근수상수이며 20을 만드는데 산수대입연산자, 19을 만드는데 감소연산자를 사용하시오.

4. 리수복영웅의 시를 출력하는 프로그람을 작성하시오. 적당한 확장문자를 사용하여 행을 바꾸시오.

5. 서고함수 islower()는 하나의 문자를 인수로 가지며 문자가 소문자이면 0아닌 옹근수, 대문자이면 0을 돌려준다. islower()함수는 머리부파일 cctype.h를 요구한다. 사용자가 문자를 입력하면 그 문자가 소문자인가 대문자인가에 따라서 0 또는 0아닌 값을 표시하는 프로그람을 작성하시오.

6. 거리를 표시하는 1mile은 1.852m, 1in은 0.0254m이다. 거리를 m로 입력하면 그것을 mile과 in으로 출력하는 프로그람을 작성하시오.

7. 섭씨온도에 9/5를 곱하고 32를 더하여 화씨온도로 변환할수 있다. 사용자가 섭

씨온도를 류동소수점수로 입력하면 그에 대응하는 화씨온도를 표시하는 프로그람을 작성하시오.

8. setw()에 의해 지정한 마당보다 값이 작을 때 수자가 없는 빈 자리들을 공백으로 채운다. 조작자 setfill()은 하나의 문자를 인수로 가지며 마당의 빈자리에 이 문자를 채운다. 지역이름과 인구수사이의 빈자리들에 공백대신 점을 채우도록 실례 2-7의 프로그람을 변경하시오. 즉

xx시………2425785

9. 두개의 분수 a/b와 c/d가 있다면 그 합은 다음식에 의하여 계산한다.

$$\frac{a}{b}+\frac{c}{d}=\frac{a\times d+c\times b}{b\times d}$$

실례로 1/4+2/3은

$$\frac{1}{4}+\frac{2}{3}=\frac{1\times 3+4\times 2}{4\times 3}=\frac{3+8}{12}=\frac{11}{12}$$

사용자가 두개의 분수를 입력하면 그 합을 분수형식으로 표시하는 프로그람을 작성하시오. 사용자는 프로그람과 다음과 같이 대화한다.

첫째 분수를 입력하시오: 1/2

둘째 분수를 입력하시오: 2/5

합 = 9/10

발취연산자를 한개이상의 량을 련속 입력하는데 사용할수 있다.

10. 토지의 면적을 계산할 때 정보, 평이라는 단위를 사용한다. 정보와 평수를 입력하면 표준단위인 ㎢와 ㎡로 표시하는 프로그람을 작성하시오. 정보수는 옹근수, 평수는 류동소수점수로 하시오.

11. 표준상태에서 출력은 마당에 오른쪽으로 맞추어 채워넣는다. 조작자 setiosflags(ios::left)를 사용하여 마당에 왼쪽으로 맞추어 본문을 출력할수 있다. setw()와 이 조작자를 사용하여 다음과 같이 출력하는 프로그람을 작성하시오.

이름	거리이름	구역이름
김인철	광복거리 칠골 1동 10반	만경대구역
리순화	통일거리 충성동 12반	락랑구역

12. 문제 10과 반대로 ㎢수와 ㎡수를 입력하면 그것을 정보와 평으로 표시하는 프로그람을 작성하시오. ㎢수는 옹근수, ㎡수는 류동소수점수형으로 하시오.

제 3 장. 순환과 분기

많은 프로그람에서는 명령문들을 처음부터 마지막까지 정확히 순차적으로 실행할 수 없다. 대부분의 프로그람은 사람들처럼 변화하는 환경에 맞게 자기의 동작을 결정한다. 조종의 흐름은 프로그람에서 수행한 계산에 기초하여 프로그람의 한 부분에서 다른 부분으로 넘어간다. 이와 같은 이행(jump)을 일으키는 프로그람명령문을 조종명령문(control statement)이라고 한다. 여기에는 두 부류 즉 순환(loop)과 분기(decision)가 있다.

이 장에서는 비교연산자, for, while, do순환, if와 if…else명령문, switch명령문, 조건연산자, 론리연산자 등에 대하여 설명한다.

제 1 절. 순환

순환을 몇번 실행하는가 혹은 코드의 어느 부분에서 분기하는가 하는것은 어떤 식이 참인가, 거짓인가 하는데 의존한다. 보통 이 식들에는 비교연산자(relational operator)가 포함되여있다. 비교연산자는 두개의 값을 비교한다. 순환과 분기조작은 비교연산자와 밀접히 련관되여있으므로 이 연산자부터 먼저 설명한다.

1. 비교연산자

비교연산자는 두개의 값을 비교한다. 이 값들은 char, int, float와 같은 C++의 기본자료형(built-in data type) 또는 사용자정의클라스(user-defined class)이다. 비교연산자에는 같기, 작기, 크기 등 여러가지가 있다. 비교결과는 참 또는 거짓이다. 실례로 두개의 값이 같을 때는 참, 같지 않을 때는 거짓이다.

실례 3-1은 비교연산자를 사용하여 옹근수변수와 상수를 비교한다.

(실례 3-1) 비교연산

```
#include <iostream>
using namespace std;
int main()
{
   int numb;
   cout << "수를 입력하십시오:";
   cin >> numb;
   cout << "numb < 10 =" << (numb < 10) << endl;
   cout << "numb > 10 =" << (numb > 10) << endl;
   cout << "numb == 10 =" << (numb == 10) << endl;
   return 0;
}
```

이 프로그람은 사용자가 입력한 수값을 10과 비교한다. 사용자가 20을 입력했을

때 출력은 다음과 같다.

```
수를 입력하십시오: 20
numb < 10 = 0
numb > 10 = 1
numb == 10 = 0
```

numb가 10보다 작으면 처음식은 참이다. numb가 10보다 크면 둘째 식이 참이고 numb가 10과 같으면 셋째 식이 참이다. 출력으로부터 알수 있는것처럼 C++번역프로그람에서 참인 식은 값 1을 가지고 거짓식은 값 0을 가진다.

표준 C++에는 bool형이 있다. bool형은 두개 상수값 true 혹은 false중 어느하나를 가진다. numb < 10과 같은 비교식의 결과는 bool형이고 프로그람은 0대신에 false, 1대신 true를 출력할수도 있다. 비교연산결과 혹은 bool형변수값을 cout <<에 의하여 출력한다면 false나 true가 아니라 0 또는 1을 출력한다. 이것은 초기에 C++에 bool형이 없었던것과 관련되여있다.

표준 C++가 출현하기 전에 거짓과 참을 표시하는 유일한 방법은 0과 1이였다. 거짓은 bool형의 false값으로 표시되거나 옹근수 0으로 표시된다. 참은 bool형의 true 또는 옹근수 1로 표시된다.

일반적으로 참 혹은 거짓값을 표시할 필요가 없으며 순환과 분기에서 프로그람이 다음에 할 일을 결정하는데 사용한다.

표 3-1에 C++비교연산자가 있다.

표 3-1. 비교연산자

연산자	의미
>	크기
<	작기
==	같기
!=	같지 않기
>=	크거나 같기
<=	작거나 같기

그러면 비교연산자를 사용하는 식들과 매개 식의 값을 고찰하자.

```
jane = 44;            // 대입명령문
harry = 12;           // 대입명령문
(jane == hary)        // 거짓
(harry <= 12)         // 거짓
(jane > harry)        // 참
(jane >= 44)          // 참
(harry != 12)         // 거짓
(7 < harry)           // 참
(0)                   // 거짓
(44)                  // 참
```

처음 두 행은 변수 harry와 jane의 값을 설정하는 대입명령문이다.

여기서 갈기연산자 ==는 갈기기호 두개를 사용한다. 일반적으로 대입연산자인 한개의 갈기기호를 비교연산자로 잘못 사용하여 오유가 발생하는 경우가 많다. 이것은 번역프로그람이 어떤 오유인지 알아낼수 없는 아주 난처한 오유이다.

C++는 1이 참을 가리키도록 만들어졌지만 0아닌 임의의 값(-7 혹은 44)도 참이라고 가정한다. 즉 0만이 false이다. 따라서 앞에서 본 실례에서 마지막 식은 참이다.

2. for순환

순환은 프로그람의 어떤 부분을 일정한 회수만큼 반복실행한다. 반복(repeatation)은 조건이 참인동안 계속된다. 조건이 거짓으로 될 때 순환은 끝나고 조종은 순환뒤의 명령문으로 넘어간다.

C++의 순환에는 세가지 즉 for순환, while순환, do순환이 있다.

for순환은 C++에서 처음으로 받아들인 순환으로서 모든 순환조종요소들이 한곳에 집중되여있다.

for순환은 코드의 한 부분을 일정한 회수만큼 반복실행한다. 보통 for는 순환에 들어가기 전에 그 코드를 반복실행해야 할 회수를 미리 아는 경우에 사용한다.

실례 3-2는 0～14까지의 수의 두제곱을 표시한다.

(실례 3-2) 단순한 for순환

```
#include <iostream>
using namespace std;
int main()
{
  int j;                        // 순환변수를 정의한다
  for(j=0; j<15; j++)       // 0부터 14까지 순환한다
     cout << j * j << " ";  // j의 2제곱을 표시한다
  cout << endl;
  return 0;
}
```

이 출력결과는 다음과 같다.

```
1  4  9  16  25  36  49  64  81  100  121  144  169  196
```

이 프로그람에 대하여 고찰하자.

for명령문은 순환을 조종한다. for순환은 예약어 for와 그뒤의 괄호안에서 반두점으로 구분된 세개의 식들로 이루어진다. 즉

```
for(j=0; j<15; j++)
```

세개의 식은 초기화식, 조건식, 증분식이다. (그림 3-1)

```
    초기화식   조건식   증분식
for ( j = 0;   j < 15;   ++j ) ( )← 반두점이 없다
    명령문;  ———— 단일명령문 순환본체
ㄱ)

    초기화식   조건식   증분식
for ( count = 0; count < 10; count ++ ) ( )← 반두점이 없다
{
    명령문;
    명령문;    복합명령문 순환본체
    명령문;
} ( )
  반두점이 없다
ㄴ)
```

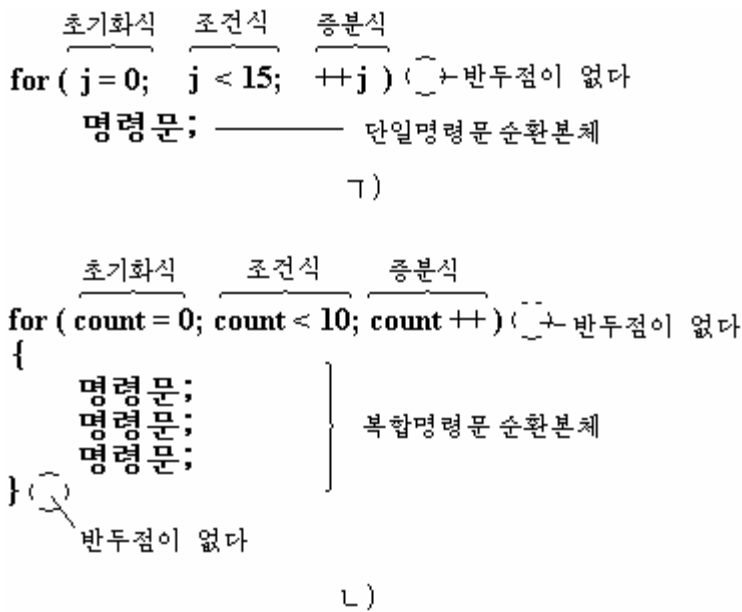



그림 3-1. for명령문의 문법

일반적으로 세개의 식에는 같은 변수가 포함되는데 이 변수를 순환변수라고 한다. 그러나 순환변수가 항상 포함되는것은 아니다. 실례 3-2에서 순환변수는 j이다. 순환변수는 순환본체안의 명령문들이 실행을 시작하기 전에 정의된다. 순환본체는 순환할때마다 실행하는 코드이다. 이 코드의 반복이 바로 순환이 존재하는 리유로 된다. 실례에서 순환본체는 단일명령문으로 이루어진다. 즉

```
cout << j * j << " ";
```

이 명령문은 j의 두제곱과 두개의 공백을 출력한다. 두제곱은 j를 두번 곱하여 얻는다. 순환할 때 j는 0, 1, 2, 3, …, 14로 증가된다. 따라서 이 수들의 두제곱이 차례로 표시된다. 즉 0, 1, 4, 9, …, 196.

for명령문의 뒤에 반두점을 쓰지 말아야 한다. 반두점을 쓰지 않으면 for명령문과 순환본체가 하나의 명령문으로 간주된다. for명령문뒤에 반두점을 쓰면 번역프로그람은 순환본체가 없는것으로 보며 프로그람은 제대로 동작하지 않는다.

for명령문안의 세개의 식이 순환을 어떻게 조종하는가를 고찰해보자.

- 초기화식

초기화식(initialization expression)은 순환을 처음 시작할 때 한번만 실행된다. 초기화식은 순환변수에 초기값을 준다. 실례 3-2에서 초기화식은 j를 0으로 설정한다.

- 조건식

보통 조건식에는 비교연산자가 들어있다. 조건식(test expression)은 매번 순환할때마다 순환본체를 실행하기 전에 평가된다. 조건식은 순환을 다시 시작하는가 끝내는가를 결정한다. 조건식이 참이라면 한번이상 순환하며 조건식이 거짓이면 순환은 끝나고 조종은 순환의 뒤에 있는 명령문에로 넘어간다. 실례 3-2에서 명령문

```
cout << endl;
```
은 순환이 끝난 다음에 실행된다.

- 증분식

증분식(increment expression)은 순환변수값을 증가시킴으로써 그 값을 변경한다. 증분식은 항상 순환의 끝에서 순환본체가 실행된 다음에 실행된다. 여기서 증가연산자 ++는 순환할 때마다 j에 1을 더한다. 그림 3-2는 for순환조작의 흐름을 보여준다.

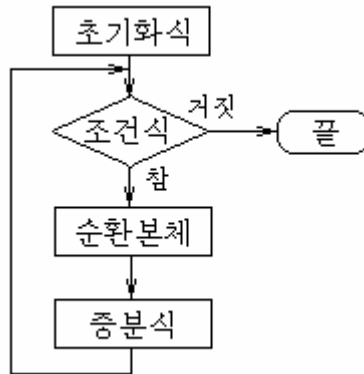



그림 3-2. for순환조작

- 순환회수

실례 3-2에서 순환회수는 정확히 15이다. 처음에 j는 초기화식에서 0으로 설정된다. 마지막으로 순환할 때 j는 조건식 j<15에 의하여 14로 결정된다. j가 15일 때 순환은 끝나고 순환본체는 실행되지 않는다. 일반적으로 이런 수들의 배렬은 어떤 코드를 일정한 회수만큼 실행하는데 사용된다. 즉 0으로부터 시작하고 작기기호를 가진 조건식안의 반복회수와 같은 값을 사용하며 매번 순환이 끝날 때마다 증가된다.

아래와 같은 for순환실례를 보기로 하자.

```
for(count=0; count<15; count++)
    // 순환본체
```
여기서 순환본체는 정확히 100번 즉 count는 0～99이다.

- 순환본체안에 있는 복합명령문

보통 순환본체안에서 한개이상의 명령문을 실행할수 있다. 복합명령문은 함수처럼 대괄호에 의해 구분된다. 순환본체의 닫긴 대괄호뒤에는 반두점이 없다. 그러나 순환본체안의 매개 명령문뒤에는 반두점이 있다. 실례 3-3은 순환본체에서 세개의 명령문을 실행한다. 이 실례는 수들의 3제곱을 2렬로 표시한다.

(실례 3-3) 1～10의 3제곱

```
#include <iostream>
#include <iomanip>                              // setw용
using namespace std;
int main()
```

```
{
  int numb;                                  // 순환변수를 정의한다
  for(numb=1; numb<=10; numb++)              // 1~10까지 순환한다
   {
     cout << setw(4) << numb;                // 첫째 렬을 표시한다
     int cube = numb * numb * numb;          // 3제곱계산
     cout << setw(6) << cube << endl;        // 둘째 렬을 표시한다
  }
  return 0;
}
```

이 프로그람의 출력은 다음과 같다.

```
  1     1
  2     8
  3    27
  4    64
  5   125
  6   216
  7   343
  8   512
  9   729
 10  1000
```

실례 3-10에서 순환변수는 0이 아니라 1로 초기화되므로 <=연산자에 의하여 9가 아니라 10으로 끝난다. 따라서 순환본체는 10번 실행되고 순환변수는 0~9가 아니라 1~10으로 된다.

- 블로크와 변수의 보임성

여러개의 명령문을 포함하는 괄호로 이루어진 순환본체를 코드블로크(code block)라고 한다. 블로크의 중요성의 하나는 블로크안에서 정의된 변수를 블로크밖에서 볼수 없다는데 있다. 보임성(visibility)은 프로그람명령문들이 어떤 변수를 호출하거나 볼수 있는가 하는것이다. 실례 3-3에서는 블로크안에서 cube변수를 정의한다. 즉

```
int cube = numb * numb * numb;
```

이 변수는 블로크밖에서 볼수 없고 괄호안에서만 볼수 있다. 그러므로 순환본체뒤에 다음과 같이 명령문을 배치하면

```
cube = 10;
```

번역프로그람은 변수 cube가 순환밖에서 정의되지 않았으므로 오유를 통보한다.

변수의 보임성을 제한하는 방법의 한가지 우점은 같은 프로그람안의 다른 블로크에서 이러한 변수를 사용할수 없다는데 있다. 블로크안에서 변수정의는 C++에서는 가능하지만 C에서는 불가능하다.

- 들여쓰기와 순환의 형태

순환본체를 오른쪽으로 들여쓰면 프로그람은 보기 편리해진다. 이렇게 하면 순환본체가 어디서 시작되고 어디서 끝나는가를 쉽게 알수 있다.

프로그람을 들여쓰는 방법에는 여러가지가 있다. 하나는 열린 괄호를 순환문에 올려놓고 본체의 다른 부분과 닫긴 괄호를 들여쓰는것이다.

```
for(numb=1; numb <= 10; numb++) {
  cout << setw(4) << numb;
  int cube = numb * numb * numb;
  cout << setw(6) << cube << endl;
}
```

이러한 형태는 행은 하나 줄어들지만 리해하기 힘들다. 왜냐하면 열린 괄호에 대응하는 닫긴 괄호를 찾기 힘들기때문이다.

또 다른 방법은 괄호가 아니라 본체를 들여쓰는것이다.

```
for(numb=1; numb <= 10; numb++)
 {
   cout << setw(4) << numb;
   int cube = numb * numb * numb;
   cout << setw(6) << cube << endl;
 }
```

- 걸음식오유수정

번역프로그람에 갖추어진 오유수정기능에는 한걸음씩 실행하는 기능이 있다.

오유수정하려는 프로그람의 프로젝트를 열고 원천파일이 포함된 편집창문을 여는것으로부터 시작한다. 오유수정프로그람을 기동시키는데 필요한 지령은 번역프로그람마다 다르다. 일정한 기능건을 눌러서 한번에 프로그람의 한 행을 실행할수 있다. 프로그람이 전진하면 실행된 명령문들의 렬이 표시된다. 순환에서는 실행된 순환안의 명령문들이 표시되고 그다음 조종은 순환의 시작으로 되돌아오며 이 주기를 계속 반복한다.

또한 프로그람을 한걸음 실행했을 때 변수값들에 어떤 변화가 생기는가를 감시하는데 오유수정프로그람을 사용할수 있다. 이것은 프로그람의 오유를 수정하기 위한 강력한 도구이다. 오유수정프로그람의 감시창문안에서 numb와 cube변수들의 변화과정을 고찰하여 실례 3-3의 프로그람을 검사할수 있다.

한걸음실행과 감시창문은 강력한 오유수정도구이다. 프로그람이 제대로 동작하지 않으면 이 기능을 리용하여 프로그람을 한걸음씩 실행하면서 주요 변수값들을 감시할수 있다.

- for순환의 사용방법

for순환에서 증분식은 순환변수를 증가시키기 위해서만 필요한것이 아니며 다른 조작을 할수도 있다.

실례 3-4에서 증분식은 순환변수를 감소시킨다. 실례 3-4는 사용자가 수값을 입력하면 그 수의 차례곱을 계산한다. 차례곱은 원래 수에 그보다 작은 정의 옹근수들을 모두 곱하여 계산한다. 즉 5의 차례곱은 5×4×3×2×1=120이다.

(실례 3-4) for순환을 사용한 차례곱계산

```
#include <iostream>
using namespace std;
int main()
{
   unsigned int numb;
   unsigned long fact=1;
   cout << "수를 입력하십시오:";
   cin >> numb;
   for(int j=numb; j>0; j--)
      fact *= j;
   cout << "차례곱=" << fact << endl;
   return 0;
}
```

이 실례에서 초기화식은 j를 사용자가 입력한 값으로 설정한다. 조건식은 j가 0보다 큰동안 순환하게 한다. 증분식은 본체를 반복실행할 때마다 j를 감소시킨다.

작은 수의 차례곱도 대단히 커지므로 unsigned long형을 차례곱값의 보관에 사용한다. Windows 98과 같은 32bit체계들에서 long은 int와 같지만 16bit체계에서는 int보다 더 크다.

다음의 출력은 작은 수의 차례곱이 얼마나 큰가를 보여준다.

```
수를 입력하십시오: 10
차례곱= 3628800
```

입력에 사용할수 있는 최대수는 12이다. 더 큰 수를 입력하면 오유통보는 없지만 결과는 long형의 범위를 초과하므로 오유로 된다.

- for명령문에서 변수의 정의

또한 실례 3-4에서는 순환변수 j를 for명령문안에서 정의한다.

```
for(int j=numb; j>0; j--)
```

이것은 C++에서 보편적인 구조로서 순환변수의 정의를 그것을 사용하는 코드에 더욱더 접근시킨다. 이와 같이 순환명령문안에서 정의한 변수들은 목록안의 정의위치로부터 목록의 끝까지의 범위에서 보임성을 가진다. 그러므로 순환밖에서도 사용하는 경우에 이 형식이 제일 좋은 수법으로 되지 않는다.

- 다중초기화식과 조건식

for명령문의 초기화부분에 여러개의 식을 반점으로 구분하여 쓸수 있다. 또한 증분식을 한개이상 쓸수 있다. 조건식은 오직 한개만 써야 한다. 실례로

```
for (j=0, alpha=100; j>50; j++, beta--)
{
   // 순환본체
}
```

이 실례는 표준순환변수 j를 가지는것과 함께 다른 변수 alpha를 초기화하며 세번째 부분에서 beta를 감소시킨다. 다중초기화식과 다중증분식은 반점으로 구분한다.

또한 for순환에서 일부 식 혹은 모든 식을 생략할수 있다. 식

```
for(; ;)
```
은 조건식이 true인 while순환과 같다. 그러나 다중식이나 식의 생략은 될수록 피해야 순환명령문의 복잡성을 없애고 쉽게 읽고 리해할수 있다.

3. while순환

for순환은 일정한 회수만큼 어떤 조작을 반복 수행한다.

그러면 순환을 시작하기 전에 반복해야 할 순환회수를 모를 때에는 어떻게 하겠는가?

이 경우에 다른 종류의 순환 즉 while순환을 사용한다.

실례 3-5는 건반으로부터 일련의 수들을 읽어들이며 0을 입력하면 순환을 끝낸다. 여기서는 0이 입력되기 전에 몇개의 수를 입력하겠는지 알수 없다.

(실례 3-5) while순환

```
#include <iostream>
using namespace std;
int main()
{
    int n=99;
    while(n != 0)
        cin >> n;
    cout << endl;
    return 0;
}
```

사용자가 0을 입력할 때까지 순환은 계속된다. 즉

```
1
27
33
144
9
0
```

while순환은 for순환의 간략판처럼 보인다. 이 순환에는 조건식이 있으나 초기화식과 증분식은 없다. 그림 3-3은 while순환의 구조를 보여준다.

그림 3-3. while순환의 구조

조건식이 참인 동안 순환은 계속된다. 실례 3-5에서 조건식

n != 0

은 사용자가 0을 입력하지 않을 때 참이다.

그림 3-4는 while순환의 조작을 보여준다. 여기서 while순환은 아주 간단하다. 초기화식이 없어도 순환변수(실례 3-5에서 n)는 순환이 시작되기 전에 초기화되여야 한다. 또한 순환본체는 순환변수값을 변화시키는 명령문을 포함하여야 하며 그렇지 않으면 순환은 끝나지 않는다.

실례 3-5에서 그것은 cin >> n;이다.

그림 3-4. while순환의 조작

- while순환에서 복합명령문

실례 3-6에서는 while순환안에서 복합명령문을 사용한다. 이것은 for순환이 있는 실례 3-3을 변경한것이다. 그러나 옹근수렬의 3제곱대신 4제곱을 계산한다. 이 프로그람에서 결과를 4자리까지 표시하려고 한다고 하자. 그러면 결과가 9999보다 커지기 전에 순환을 중지해야 한다.

(실례 3-6) 4제곱

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
  int pow = 1;
  int numb = 1;
  while(pow < 9999)
  {
     cout << setw(2) << numb;
     cout << setw(5) << pow << endl;
     ++numb;
     pow = numb * numb * numb * numb;
  }
  cout << endl;
  return 0;
}
```

numb의 4제곱을 얻기 위하여 numb자체를 4번 곱한다. numb는 순환할 때마다 증가된다. 그러나 while의 조건식에서는 numb를 사용하지 않는다. 그대신에 pow의 결과값은 순환이 끝날 때 결정된다. 그러므로 출력은 다음과 같다.

```
1     1
2    16
3    81
4   256
5   625
6  1296
7  2401
8  4096
9  6561
```

- 산수연산자와 비교연산자의 우선순위

실례 3-7은 연산자의 우선순위를 보여준다. 이 프로그람은 피보나씨수렬(Fibonacci series)이라고 부르는 수들의 렬을 만든다. 아래에서 수렬의 처음 여러항을 보여준다.

```
0  1  2  3  5  8  13  21  34  55
```

매개 항은 앞에 있는 두개 항의 값들을 더하여 얻는다. 즉 1+1은 2, 1+2는 3, 2+3은 5, 3+5는 8,….

(실례 3-7) while순환에 의한 피보나씨수렬의 계산

```
#include <iostream>
using namespace std;
int main()
{
  const unsigned long limit = 4294967295;
  unsigned long next = 0;
  unsigned long last = 1;
```

```
    while(next < limit / 2)
    {
        cout << last << " ";
        long sum = next + last;
        next = last;
        last = sum;
    }
    cout << endl;
    return 0;
}
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711…
```

실례 3-7에서는 가장 큰 정의 옹근수를 보관하는 unsigned long형변수를 사용한다. while명령문의 조건식은 unsigned long형의 제한값을 초과하는 수의 바로 앞에서 순환을 끝낸다. 이 제한값을 const로 정의하면 그 값을 변경할수 없다. next가 제한값의 절반보다 커지면 중지한다. 그렇지 않으면 sum은 제한값을 초과한다.

조건식은 두개 연산자를 사용한다.

```
(next < limit / 2)
```

목적은 next와 limit/2의 결과를 비교하는것이다. 즉 비교하기 전에 나누기를 진행해야 한다. 나누기를 먼저 수행하도록 하기 위하여 나누기를 괄호안에 넣을수 있다.

```
(next < (limit / 2) )
```

그러나 괄호는 필요없다. 왜냐하면 나누기연산자의 우선순위가 비교연산자보다 높기때문이다. 따라서 괄호가 없어도 limit/2는 비교하기 전에 평가된다.

4. do순환

while순환에서 조건식은 순환의 시작에서 평가되므로 순환에 들어갈 때 조건식이 거짓이면 순환본체는 한번도 실행되지 않는다. 어떤 경우에는 그것이 필요하지만 때때로 조건식의 초기상태에 관계없이 순환본체가 적어도 한번은 실행되여야 한다. 이것은 do순환에서 가능하다. do순환에서는 조건식을 순환의 끝에 놓는다.

실례 3-8에서는 사용자가 나누이는 수와 나누는 수를 입력하면 /와 %연산자를 사용하여 상과 나머지를 계산하고 그 결과를 출력한다.

(실례 3-8) do순환

```
#include <iostream>
using namespace std;
int main()
{
    long dividend, divisor;
    char ch;
    do
    {
        cout << "나누이는 수를 입력하시오:";
        cin >> dividend;
```

```
        cout << "나누는 수를 입력하시오:";
        cin >> divisor;
        cout << "상=" << dividend / divisor;
        cout << ", 나머지=" << dividend % divisor;
        cout << "\n 계속하겠습니까(y/n) ?:";
        cin >> ch;
    } while(ch != 'n');
    return 0;
}
```

이 프로그람의 대부분은 do순환안에 있다. 우선 예약어 do는 순환의 시작을 표시한다. 마지막에 있는 while명령문은 조건식을 제공하며 순환을 끝낸다. do순환의 while명령문은 순환의 끝이 반두점으로 끝난다는것을 제외하면 while순환과 거의 같다.

그림 3-5에서 do순환의 구조를 보여주었다.

그림 3-5. do순환의 구조

실례 3-8은 순환본체를 실행할 때마다 사용자에게 다시 반복하겠는가를 묻는다. 사용자는 다시 반복하기 위하여 'y'문자를 입력하면 조건식

ch != 'n'

이 참으로 된다. 사용자가 'n'을 입력하면 조건식은 거짓으로 되고 순환은 끝난다. 그림 3-6은 do순환의 조작을 보여준다.

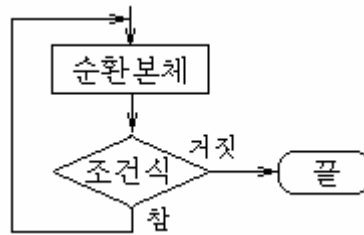



그림 3-6. do순환의 조작

프로그람의 출력은 다음과 같다.

```
나누이는 수를 입력하시오: 11
나누는 수를 입력하시오: 3
상= 3, 나머지= 2
계속하겠습니까(y/n) ?: y
나누이는 수를 입력하시오: 222
나누는 수를 입력하시오: 17
상= 13, 나머지= 1
계속하겠습니까(y/n) ?: n
```

- 순환의 사용법

지금까지 세가지 순환명령문을 고찰하였다. for순환은 반복회수를 미리 알 때 적합하다. while과 do순환은 반복회수를 모르는 경우에 사용된다. while순환은 순환본체를 한번도 실행하지 않는 경우가 있을 때 사용되고 do순환은 순환본체를 적어도 한번 실행해야 할 때 사용된다.

제 2 절. 분기

프로그람에서 분기(decision)가 요구되는 경우도 있다. 분기는 식의 결과에 따라서 프로그람의 서로 다른 부분에로 이행하게 한다.

C++에서 분기는 여러가지 방법으로 얻을수 있다. 가장 중요한 방법은 if～else명령문으로서 두가지중 하나를 선택하는것이다. 이 명령문은 else없이 단순 if명령문으로 사용할수도 있다. 또 하나의 분기명령문으로서 switch는 단일변수값에 따라서 코드의 여러 부분에로 분기할수 있다. 특수한 경우에는 조건연산자를 사용한다.

1. if명령문

if명령문은 가장 간단한 분기명령문이다. 실례 3-9는 그 실례이다.

(실례 3-9) if명령문

```
#include <iostream>
using namespace std;
int main()
{
```

```
    int x;
    cout << "수를 입력하십시오:";
    cin >> x;
    if(x > 100)
        cout << "이 수는 100보다 큽니다.\n";
    return 0;
}
```
조건식은 if예약어뒤의 괄호안에 쓴다. if명령문의 구조를 그림 3-7에 주었다.

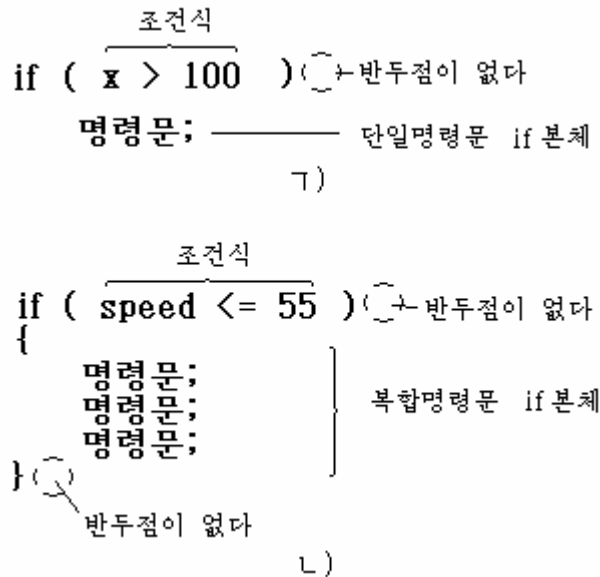



그림 3-7. if명령문의 구조

if의 구조는 while의 구조와 비슷하다. 차이는 if의 뒤에 오는 명령문은 조건식이 참인 경우에 한번만 실행되지만 while의 뒤에 오는 명령문은 조건식이 거짓으로 될때까지 반복실행되는것이다. 그림 3-8은 if명령문의 조작을 보여준다.

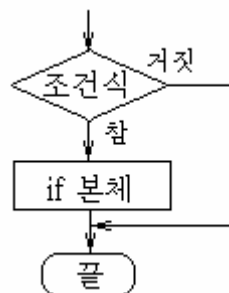



그림 3-8. if명령문의 조작

이 실례는 사용자가 100보다 큰 수를 입력했을 때 다음과 같이 출력한다.

```
수를 입력하십시오: 2000
이 수는 100보다 큽니다.
```

입력한 수값이 100보다 크면 프로그람은 두번째 행을 출력하지 않고 끝난다.

- if본체안의 복합명령문

순환과 마찬가지로 if본체안의 코드는 단일명령문으로 이루어지거나 괄호에 넣은 명령문블로크로 이루어진다. 실례 3-10은 실례 3-9를 약간 변경한것이다.

실례 3-10. 여러행 본체를 가지는 if명령문

```
#include <iostream>
using namespace std;
int main()
{
   int x;
   cout << "수를 입력하십시오:";
   cin >> x;
   if(x > 100)
   {
      cout << "수값 " << x;
      cout << "는 100보다 큽니다.\n";
   }
   return 0;
}
```

이 프로그람의 출력은 다음과 같다.

```
수를 입력하십시오: 12345
수값 12345는 100보다 큽니다.
```

- 순환안에서 if의 겹쌓임

순환과 분기구조는 겹쌓일수 있다. 순환안에 if를 겹쌓을수 있으며 if안에 순환을, if안에 if를 겹쌓을수 있다.

실례 3-11에서는 for순환안에 if를 겹쌓고있다. 이 실례는 입력한 수가 씨수인가를 결정한다. 씨수란 1과 그 수자체에 의해서만 나누어지는 옹근수이다. 실례로 1, 2, 3, 5, 7, 11, 13, 17, ….

(실례 3-11) if명령문을 사용한 씨수의 표시

```
#include <iostream>
using namespace std;
#include <process.h>
int main()
{
   unsigned long n, j;
   cout << "수를 입력하십시오:";
   cin >> n;
   for(j=2; j<=n/2; j++)
      if(n%j == 0)
      {
         cout << "이 수는 씨수가 아닙니다."
              << j << "으로 나누어집니다.\n";
         exit(0);
      }
```

```
        cout << "이 수는 씨수입니다.\n";
        return 0;
    }
```

이 실례에서 사용자는 n에 어떤 수를 입력한다. 그다음 프로그람은 for순환을 사용하여 n을 n/2까지의 모든 수로 나눈다. 나누는 수는 순환변수 j이다. n이 j의 임의의 값으로 완전히 나누일 때 n은 씨수가 아니다. 어떤 수가 다른 수로 완전히 나누이면 나머지는 0이므로 j의 매개 값에 대한 이 조건을 검사하기 위하여 if명령문안에서 나머지연산자 %를 사용한다. 그 수가 씨수가 아니면 그것을 사용자에게 알리고 프로그람을 완료한다.

이 프로그람의 출력은 다음과 같다.

```
수를 입력하십시오: 13
이 수는 씨수입니다.
수를 입력하십시오: 22229
이 수는 씨수입니다.
수를 입력하십시오: 22231
이 수는 씨수가 아닙니다. 11로 나누어집니다.
```

여기서는 순환본체의 둘레에 괄호가 없다. 이것은 if명령문과 그 본체안의 명령문들이 단일명령문으로 간주되기때문이다. 알기 쉽게 하기 위하여 괄호를 넣을수 있지만 번역프로그람은 그것을 요구하지 않는다.

- 서고함수 exit()

실례 3-11은 수가 씨수가 아닐 때 즉시 완료한다. 따라서 수가 씨수가 아니라는것을 한번밖에 검사하지 않는다. 이것은 서고함수 exit()에 의하여 달성된다. exit()함수는 프로그람을 완료하며 돌림값을 돌려주지 않는다. 실례에서는 하나의 인수 0을 프로그람을 완료할 때 조종을 위하여 돌려준다. 이 값은 exit()에 의해 제공된 돌림값을 얻어서 ERRORLEVEL값을 사용하는 묶음파일(batch file)에 필요하다. 보통 값 0은 오유없는 완료를 의미하고 다른 값들은 오유를 의미한다.

2. if～else명령문

if명령문은 조건이 참이면 무엇인가 수행하고 조건이 참이 아니면 아무것도 수행하지 않는다. 그런데 조건이 참이면 어떤 조작을 수행하고 조건이 거짓이면 또 다른 조작을 하는 경우도 있다. 이때 if～else명령문을 사용한다.

if～else명령문은 if문과 그뒤의 명령문(혹은 명령문블로크), 예약어 else와 그뒤에 오는 또 다른 명령문(혹은 명령문블로크)으로 이루어진다. 구조는 그림 3-9와 같다.

```
         조건식
if ( x > 100 )
    명령문; ———— 단일명령문 if 본체
else
    명령문; ———— 단일명령문 else본체
```

ㄱ)

```
          조건식
if ( zebra != 0 )
{
    명령문;
    명령문;        복합명령문 if 본체
}
else
{
    명령문;
    명령문;        복합명령문 else본체
}
```

ㄴ)

그림 3-9. if∼else명령문의 구조

실례 3-12는 if와 else가 있는 실례이다.

(실례 3-12) if∼else명령문

```
#include <iostream>
using namespace std;
int main()
{
   int x;
   cout << "수를 입력하십시오:";
   cin >> x;
   if(x > 100)
      cout << "이 수는 100보다 큽니다.\n";
   else
      cout << "이 수는 100보다 크지 않습니다.\n";
   return 0;
}
```

if명령문안의 조건식이 참이면 프로그람은 어떤 통보문을 출력하고 참이 아니면 다른 통보문을 출력한다. 출력결과는 다음과 같다.

```
수를 입력하십시오: 300
이 수는 100보다 큽니다.
수를 입력하십시오: 30
이 수는 100보다 크지 않습니다.
```

if～else명령문의 조작을 그림 3-10에 주었다.

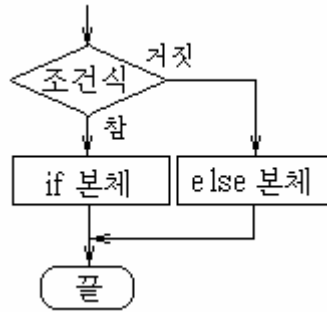



그림 3-10. if～else명령문의 조작

- getche()서고함수

실례 3-13은 while순환안에 있는 if～else명령문을 보여준다. 또한 서고함수 getche()를 사용한다. 이 프로그람은 사용자가 입력한 문자렬의 단어수와 문자수를 계산한다.

(실례 3-13) 건반입력한 문자와 단어들의 계수

```
#include <iostream>
using namespace std;
#include <conio.h>
int main()
{
   int chCount = 0;
   int wdCount = 0;
   char ch = 'a';
   cout << "영어문장을 입력하십시오:";
   while(ch != '\r')
   {
      ch = getche();
      if(ch == ' ')
          wdCount++;
      else
          chCount++;
   }
   cout << "\n단어수=" << (wdCount+1) << endl
        << "문자수=" << (chCount-1) << endl;
   return 0;
}
```

지금까지는 입력에 cin과 cout만 사용하였다. 이 수법은 입력이 끝났다는것을 알리기 위하여 사용자가 Enter건을 누를것을 항상 요구한다. 이것은 사용자가 문자를 하나 입력하고 Enter건을 눌러도 참이다. 그러나 실례 3-13과 같은 프로그람에서는 Enter건을 누를 때까지 사용자가 입력한 매개 문자를 처리해야 한다. getche()서고함수는 이러한 동작을 한다. getche()함수는 입력하자마자 그 문자를 돌려주고 인수를 가지지 않으며 CONIO.H머리부파일을 요구한다. 실례 3-13에서 getche()함수가 돌려보낸 문

자의 값은 ch에 대입된다. (getche()함수는 화면에 문자를 출력한다. getche의 끝에 있는 문자 e는 echo를 의미한다. 다른 함수 getch()는 getche()와 비슷하지만 화면에 문자를 출력하지 않는다.)

if～else명령문은 문자가 공백이면 단어수 wdCount를 증가시키고 문자가 공백이 아닌 다른 문자이면 chCount를 증가시킨다. 즉 공백이 아닌 문자를 계수한다. (이 프로그람은 아주 단순하며 단어들사이의 공백에 의하여 동작한다.)

실행결과는 다음과 같다.

```
영어문장을 입력하십시오: For while and do
단어수= 4
문자수= 13
```

while문에서 조건식은 Enter건을 누를 때 건반으로부터 받아들인 문자 ch가 '\r'문자와 같은가를 검사하고 같으면 순환과 프로그람을 완료한다.

- 대입식

실례 3-13에서 대입식과 그 우선순위를 알수 있도록 한개 코드행을 고쳐쓰자. 구조는 복잡하지만 C++와 C에서 많이 사용된다.

(실례 3-14) 건반입력한 문자와 단어들의 계수

```
#include <iostream>
using namespace std;
#include <conio.h>
int main()
{
  int chCount = 0, wdCount = 1;
  char ch;
  while((ch = getche()) != '\r')
  {
    if(ch == ' ')
        wdCount++;
    else
        chCount++;
  }
  cout << "\n단어수=" << wdCount << endl
       << "문자수=" << chCount << endl;
  return 0;
}
```

getche()가 돌려준 값은 앞의 실례처럼 ch에 대입되지만 전체 대입식은 while의 검사식안으로 옮겨졌다. 순환이 완료하는가를 알아보기 위하여 대입식을 '\r'와 비교한다. 이것은 전체 대입식이 대입에서 사용한 값을 가지게 되므로 정확히 동작한다. 즉 getche()가 'a'를 돌려주면 ch는 값 'a'를 가질뿐아니라 식(ch = getche())도 역시 값 'a'를 가진다. 이 값은 그다음 '\r'와 비교된다.

대입식은 값을 가지므로 다음과 같은 명령문을 쓸수 있다.

```
x = y = z = 0;
```

C++에서 이런 식은 옳다. 우선 z가 값 0을 가지고 그다음 z=0이 값 0을 가지게 되고 그것이 y에 대입된다. 그다음 식 y=z=0이 마찬가지로 값 0을 가지게 되고 그것이 x에 대입된다.

대입연산자 =의 우선순위가 비교연산자 !=보다 낮으므로 식 (ch = getche())에서 대입식의 둘레에 괄호를 사용한다. 괄호가 없으면 식은 다음과 같이 평가된다.

```
while(ch = (getche() != '\r'))
```

이것은 ch에 참 혹은 거짓값을 대입하게 한다.

실례에서 while명령문은 적은 공간으로 많은 능력을 제공해준다. 이것은 검사식(ch가 '\r'인가를 검사)일뿐아니라 건반으로부터 문자를 얻어서 ch에 대입한다.

- 겹쌓인 if～else명령문

실례 3-15는 보물찾기놀이의 한 장면을 보여준다.

(실례 3-15) 보물찾기(if-else사용)

```
#include <iostream>
using namespace std;
#include <conio.h>
int main()
{
  char dir = 'a';
  int x = 10, y = 10;
  cout << "중지하려면 Enter건을 누르시오.\n";
  while(dir != '\r')
  {
     cout << "\n동무의 위치는 " << x << "," << y;
     cout << "\n방향건(북-n, 남-s, 동-e, 서-w)을 누르시오:";
     dir = getche();
     if(dir == 'n')
         y--;
     else
         if(dir == 's')
             y++;
         else
             if(dir == 'e')
                 x++;
             else
                 if(dir == 'w')
                     x--;
  }
  cout << endl;
  return 0;
}
```

프로그람을 실행하면 무연한 들판에서 자기의 위치를 찾는다. 사용자는 프로그람에 어느 방향으로 가겠는가를 지정한다. 한편 프로그람은 사용자의 위치를 기억하고 사용자가 이동할 때마다 그의 위치를 돌려준다.

프로그람의 실행결과는 다음과 같다.

```
동무의 위치는 10, 10
방향건(북-n, 남-s, 동-e, 서-w)을 누르시오:n
동무의 위치는 10, 9
방향건(북-n, 남-s, 동-e, 서-w)을 누르시오:s
동무의 위치는 11, 9
방향건(북-n, 남-s, 동-e, 서-w)을 누르시오:
```

프로그람의 실행을 중지하려면 Enter건을 누른다.

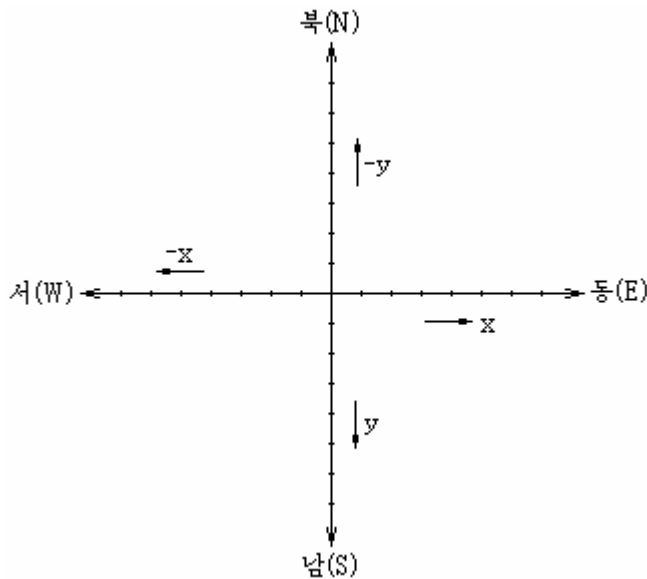



그림 3-11. 실례 3-15

이 프로그람은 다중분기를 조종하는 방법을 보여준다. 여기서는 if∼else안에 여러개의 if∼else를 겹쌓는다. 첫 검사조건이 거짓이면 둘째 조건을 시험하고 이와 같은 방법으로 4가지 요구를 모두 시험한다. 임의의 하나가 참으로 되면 적당한 조작 즉 x 혹은 y자리표를 변경하는 조작을 수행하고 프로그람은 모든 겹쌓인 분기로부터 완료한다. 이렇게 if∼else명령문들이 겹쌓이는 부분을 분기나무(decision tree)라고 한다.

- else에 대응하는 if

겹쌓인 if∼else명령문에서 위치문제가 있다. 잘못하면 다른 if와 짝을 이루는 else로 판단할수 있다. 다음의 실례 3-16이 있다.

(실례 3-16) 다른 if와 짝을 이루는 else

```
#include <iostream>
using namespace std;
int main()
{
   int a, b, c;
   cout << "세개의 수 a, b, c를 입력하십시오:\n";
   cin >> a >> b >> c;
```

```
    if(a == b)
        if(b == c)
            cout << "a와 b, c는 같습니다.\n";
    else
        cout << "a와 b, c는 다릅니다.\n";
    return 0;
}
```

여기서는 하나의 cin으로 여러개의 값을 입력한다. 매개 값을 입력한 다음 Enter를 누르면 세개의 값이 a, b, c에 각각 대입된다.

만일 2, 3, 3을 입력하면 어떤 일이 생기는가? 변수 a에는 2, b에는 3, c에는 3이 대입되고 첫 조건식은 거짓이므로 else로 넘어가서 a와 b는 다른 값이라고 출력한다고 생각할수 있다. 그러나 아무것도 출력되지 않는다. 왜서인가? 그것은 else가 다른 if와 짝을 잘못 이루었기때문이다. 여기서 else가 첫 if와 짝을 이루리라고 생각했는데 둘째 if와 짝을 이루고있다. else는 항상 자기의 else를 가지지 않는 마지막 if와 대응된다. 여기에 다음과 같은 수정판이 있다.

```
if(a == b)
    if(b == c)
        cout << "a와 b, c는 같습니다.\n";
else
        cout << "a와 b, c는 다릅니다.\n";
```

여기서는 들여쓰기와 else본체에 의해 출력문을 변경하였다. 2, 3, 3을 입력하면 아무것도 출력되지 않지만 2, 2, 3을 입력하면 다음과 같이 출력된다.

```
a와 b, c는 다릅니다.
```

실제로 첫째 if와 짝을 이루게 하기 위하여 바깥쪽 if에 괄호를 쓸수 있다.

```
if(a == b)
{
    if(b == c)
        cout << "a와 b, c는 같습니다.\n";
}
else
        cout << "a와 b, c는 다릅니다.\n";
```

여기서는 else가 첫 if와 짝을 이루고있다. 괄호는 그 안에서 다음에 오는 else를 볼수 없게 한다.

3. else if구조

실례 3-15에서 겹쌓인 if～else명령문은 리해하기 힘들다.

특히 if명령문이 더 깊숙히 겹쌓여진다면 더욱 더 리해하기 힘들다. 그러나 같은 명령문을 쓰는 다른 방법이 있다. 다음의 실례 3-17을 고찰하자.

(실례 3-17) 보물찾기(else if사용)

```
#include <iostream>
using namespace std;
```

```
#include <conio.h>
int main()
{
   char dir = 'a';
   int x = 10, y = 10;
   cout << "중지하려면 Enter건을 누르시오.\n";
   while(dir != '\r')
   {
      cout << "\n동무의 위치는 " << x << "," << y;
      cout << "\n방향건(북-n, 남-s, 동-e, 서-w)을 누르시오:";
      dir = getche();
      if(dir == 'n')
          y--;
      else if(dir == 's')
          y++;
      else if(dir == 'e')
          x++;
      else if(dir == 'w')
          x--;
   }
   cout << endl;
   return 0;
}
```

번역프로그람은 이것을 실례 3-15와 같은것으로 보지만 else뒤에는 if명령문이 배치된다.

따라서 새 예약어 else if로서 볼수 있다. 프로그람은 조건식이 참일 때까지 else if의 사다리를 타고 내려갈것이다. 그리고 다음 명령문을 실행하고 사다리를 완료한다. 이 형식은 if~else의 경우보다 리해하기 쉽다.

4. switch명령문

큰 규모의 분기구조가 있고 분기가 같은 변수값에 의존한다면 if~else 또는 else if의 사다리구조대신에 switch명령문을 사용할수 있다.

(실례3-18) switch명령문

```
#include <iostream>
using namespace std;
int main()
{
   int speed;
   cout << "\n33,45 혹은 78을 입력하십시오:";
   cin >> speed;
   switch(speed)
   {
   case 33:
      cout << "속도가 뜹니다.\n";
```

```
        break;
    case 45:
        cout << "속도가 보통입니다.\n";
        break;
    case 78:
        cout << "속도가 빠릅니다.\n";
        break;
    }
    return 0;
}
```

이 프로그람은 사용자가 수 33, 45, 78중 어느것을 입력하는가에 따라서 세개의 가능한 문자렬들중 하나를 출력한다. 여기서 저속은 33, 중속은 45, 고속은 78이다.

예약어 switch뒤에 있는 괄호에는 분기변수(switch variable)가 들어있다.

```
switch(speed)
```

그다음 괄호에는 case명령문들이 포함되여있다. 예약어 case뒤에는 상수가 놓이고 그 뒤에 두점이 놓인다. 즉

```
case 33:
```

case상수의 자료형은 분기변수와 일치해야 한다. 그림 3-12는 switch명령문의 구조를 보여준다. 분기에 들어가기 전에 프로그람은 분기변수에 값을 대입하여야 한다. 이 값은 보통 case예약어뒤에 있는 상수와 일치한다. 만일 일치하면 case뒤에 오는 명령문들에서 break가 나타날 때까지 모두 실행한다.

실례 3-18의 실행결과는 다음과 같다.

```
33,45 혹은 78을 입력하십시오: 45
속도가 보통입니다.
```

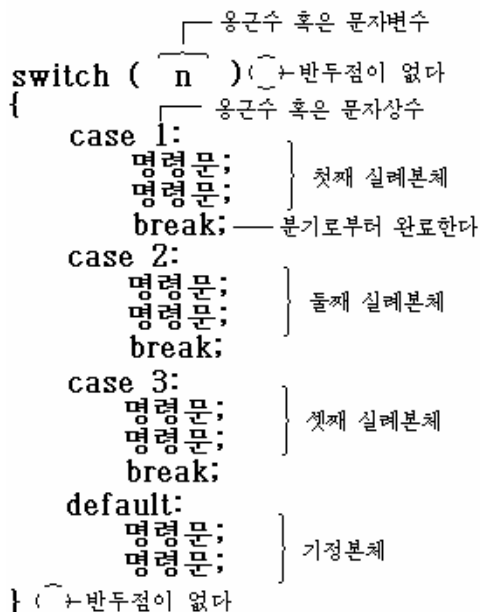



그림 3-12. switch명령문의 구조

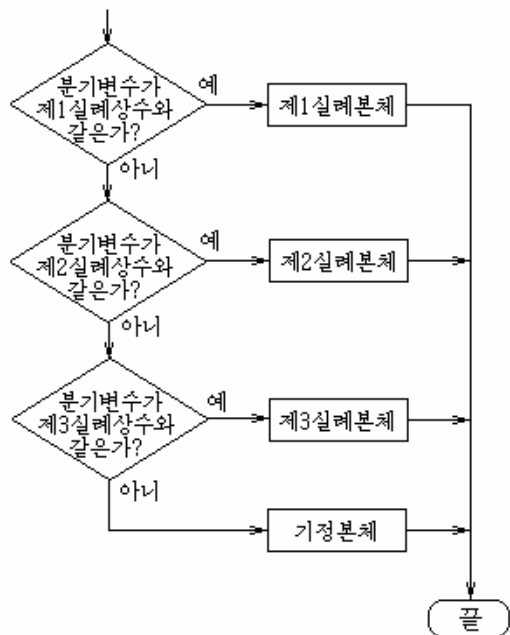



그림 3-13. switch명령문의 조작

- break명령문

실례 3-18에서는 매개 case에 break명령문이 있다. break예약어는 switch명령문을 완료시킨다. 조종은 switch구조의 다음에 놓인 첫 명령문으로 이행하며 실례 3-18에서 이것은 프로그람의 끝이다. break가 없으면 조종은 다음의 case에 해당한 명령문으로 이행한다.

분기변수의 값이 임의의 어떤 상수와도 일치하지 않으면 아무것도 실행하지 않고 조종은 분기의 끝으로 이행한다. switch명령문의 조작을 그림 3-13에 보여준다. 또한 break예약어는 순환에서 벗어날 때에도 사용된다.

- 문자변수를 가지는 switch명령문

실례 3-18에서는 int형변수에 기초한 switch명령문을 보여주었다. 실례 3-19는 char형변수에 기초한 switch명령문을 보여준다.

(실례 3-19) 보물찾기(switch사용)

```
#include <iostream>
using namespace std;
#include <conio.h>
int main()
{
   char dir = 'a';
   int x = 10, y = 10;
   cout << "중지하려면 Enter건을 누르시오.\n";
   while(dir != '\r')
   {
      cout << "\n동무의 위치는 " << x << "," << y;
      cout << "\n방향건(북-n, 남-s, 동-e, 서-w)을 누르시오:";
      dir = getche();
      switch(dir)
      {
      case 'n': y--;
          break;
      case 's': y++;
          break;
      case 'e': x++;
          break;
      case 'w': x--;
          break;
      case '\r': cout << "\n끝마칩니다.";
          break;
      default: cout << "다시 실행하십시오.\n";
      }
   }
   cout << endl;
   return 0;
}
```

문자변수 dir는 분기변수로 쓰이며 문자상수 'n', '6',…은 상수로서 사용된다. 두개의 실례에서와 같이 분기변수로서 옹근수와 문자를 사용할수 있으며 류동소수점수를 사용할수 없다.

여기서는 간단히 매 case예약어뒤에 오는 명령문을 한행에 썼다. 또한 Enter건을 누를 때 완료통보문을 출력하기 위해 case를 추가하였다.

- default예약어

실례 3-19에서는 switch구조의 맨 끝에 예약어 default가 있다. 이 예약어는 분기변수의 값이 어떤 case상수와도 일치하지 않을 때 조작하는 방법을 준다. 여기서는 사용자가 알려지지 않은 문자를 입력하였을 때 《다시 실행하십시오.》라고 출력하게 하는데 default를 사용한다. default는 항상 switch의 끝에 놓이므로 default뒤에는 break가 필요없다.

switch명령문은 사용자에 의한 입력을 해석하는 일반적인 수법이다. 가능한 매개 문자는 case에 의해 표시된다. default가 필요없는 경우에도 모든 switch명령문에서 default예약어를 사용하면 좋다.

```
default:
    cout << "오유: switch에 대한 입구가 옳지 않습니다.\n";
    break;
```

와 같은 구조는 작성자에게 프로그람의 조작에서 오유가 있다는것을 알린다.

- switch와 if～else의 비교

if～else(또는 else if)명령문과 switch명령문을 비교해보자.

if～else구조에서는 서로 련관되지 않은 변수들을 포함하는 일련의 식들을 사용할수 있다.

```
if(steamPressure * factor > 56)
    // 명령문들
else if(voltageIn + voltageOut < 2300)
    // 명령문들
else if(day == Thirsday)
    // 명령문들
else
    // 명령문들
```

그러나 switch명령문에서 모든 분기는 같은 변수에 의하여 선택된다. 즉 유일한것은 이 변수의 값이다. 즉 다음과 같이 쓸수 없다.

```
case a<3:
    // …
    break;
```

case상수는 3 혹은 'a'와 같은 옹근수 혹은 문자상수이거나 'a' + 32와 같이 상수로 평가되는 식이여야 한다.

이런 조건과 만날 때 switch명령문은 아주 간단명료하고 리해하기 쉽다.

제 3 절. 조건연산자와 론리연산자

1. 조건연산자

어떤 값이 참이면 변수에 어떤 값을 주고 그것이 거짓이면 다른 값을 주어야 하는 경우가 있다. 여기에 if～else명령문의 실례가 있다. 이 실례에서는 변수 min에 어느것이 작은가에 따라서 alpha 혹은 beta값을 준다.

```
if(alpha < beta)
   min = alpha;
else
   min = beta;
```

이러한 구조는 C++의 조건연산자에 의하여 간단히 표시할수 있다. 조건연산자(conditional operator)는 세개의 연산수에 대하여 조작하는 두개의 기호로 이루어진다. C++에는 이러한 연산자가 오직 한개뿐이며 다른 연산자들은 한개 혹은 두개의 연산수에 대하여 동작한다. 우의 코드를 조건연산자를 사용하여 다음과 같이 쓸수 있다.

```
min = (alpha < beta) ? alpha : beta;
```

같기기호의 오른변에 있는 명령문부분을 조건식이라고 한다.

```
(alpha < beta) ? alpha : beta;        // 조건식
```

물음표와 두점은 조건연산자를 이룬다. 물음표앞의 식 (alpha < beta)는 검사식이다. 검사식과 alpha, beta는 세개의 연산수이다.

검사식이 참이면 전체 조건식은 물음표뒤의 연산수 alpha의 값을 가지고 검사식이 거짓이면 조건식은 두점뒤에 오는 연산수 beta값을 가진다. 검사식에서는 괄호가 필요없지만 명령문을 읽기 쉽게 하기 위하여 괄호를 사용한다. 그림 3-14는 조건연산자의 구조를 보여주며 그림 3-15는 그 조작을 보여준다.

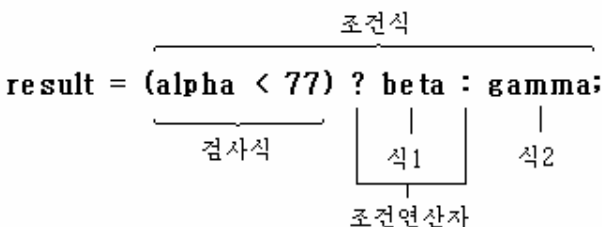
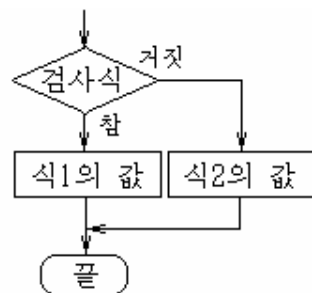



그림 3-14. 조건연산자의 구조　　　　그림 3-15. 조건연산자의 조작

조건식을 다른 변수에 대입할수 있다. 실례에서 조건식은 변수 min에 대입된다.

다른 실례를 고찰하자. 변수 n의 절대값을 얻는데 조건연산자를 사용하는 명령문이 있다.

```
absValue = n < 0 ? -n : n;
```

n이 0보다 작으면 식은 -n으로 되고 n이 0보다 작지 않으면 식은 n으로 된다. 결

과는 n의 절대값으로 된다.

실례 3-20은 조건연산자를 사용하여 1행의 본문에서 8문자폭의 마당마다 x를 하나씩 출력하므로 타브중지점이 화면우에서 어느 위치에 있는가를 알아보는데 사용할수 있다.

(실례 3-20) 8칸마다 x를 출력(조건연산자사용)

```
#include <iostream>
using namespace std;
int main()
{
   for(int j=0; j<80; j++)
   {
      char ch = (j % 8) ? ' ' : 'x';
      cout << ch;
   }
   return 0;
}
```

실행결과는 다음과 같다.

```
x       x       x       x       x       x       x       x       x       x
```

j가 0～79까지 순환할 때 식 (j % 8)이 false 즉 0으로 될 때 j는 8의 배수이다. 따라서 조건식

```
(j % 8) ?  ' ' : 'x'
```

은 j가 8의 배수가 아닐 때 값 ' '(공백문자)을 가지고 8의 배수일 때 'x'를 가진다. 순환본체안의 두개의 명령문에서 ch변수를 삭제하여 하나로 결합할수 있다. 즉

```
cout << (j % 8) ? ' ' : 'x';
```

2. 론리연산자

론리연산자(logical operator)는 론리변수(즉 true 혹은 false값을 가지는 bool형의 변수)들을 론리적으로 결합한다. 실례로 《오늘은 로동일이다.》는 론리값이므로 참 또는 거짓이다. 또한 론리식 《김동무에게는 자전거가 없다.》고 하자. 그러면 이 식들을 론리적으로 결합할수 있다. 즉 《오늘은 로동일이고 김동무에게는 자전거가 없다.》고 한다면 그는 뻐스를 타야 한다. 여기서 론리결합은 문장이며 그것은 두개 문장의 결합에 참 혹은 거짓값을 제공해준다. 따라서 두 문장이 모두 참이면 김동무는 뻐스를 탄다.

1) 론리적

C++에서 론리연산자들이 론리식을 어떻게 결합하는가를 고찰하자. 실례 3-21은 실례 3-19의 보물찾기놀이를 보기좋게 하는데 론리적연산자(logical AND)를 사용한다. 보물을 자리표 (7,11)에 묻어놓고 선수가 그것을 찾는가 지켜본다.

(실례 3-21) 론리적연산자

```
#include <iostream>
```

```
using namespace std;
#include <process.h>
#include <conio.h>
int main()
{
  char dir = 'a';
  int x = 10, y = 10;
  cout << "중지하려면 Enter건을 누르시오.\n";
  while(dir != '\r')
  {
    cout << "\n동무의 위치는 " << x << "," << y;
    cout << "\n방향건(북-n, 남-s, 동-e, 서-w)을 누르시오:";
    dir = getche();
    switch(dir)
    {
    case 'n': y--; break;
    case 's': y++; break;
    case 'e': x++; break;
    case 'w': x--; break;
    }
    if(x == 7 && y == 11)
    {
        cout << "\n보물을 찾았습니다.\n";
        exit(0);
    }
  }
  cout << endl;
  return 0;
}
```

이 프로그람에서 중요한것은 if명령문이다.

```
if(x == 7 && y == 11)
```

조건식은 x가 7이면서 y가 11이면 참이다. 론리적연산자 &&는 두개의 비교식을 결합하여 그 결과를 얻는다. 비교식이란 비교연산자를 사용한 식이다.

비교식의 주위에는 괄호가 필요없다. 즉 다음과 같이 쓰지 않아도 된다.

```
((x == 7) && (y == 11))   // 안쪽 괄호는 필요없다.
```

이것은 비교연산자의 우선순위가 론리연산자보다 높기때문이다. 실행결과는 다음과 같다.

```
동무의 위치는 7,10
방향건(북-n, 남-s, 동-e, 서-w)을 누르시오: s
보물을 찾았습니다.
```

C++에는 세개의 론리연산자가 있다. (표 3-2)

표 3-2. 론리연산자

연산자	효과
&&	론리적

연산자	효과
||	론리합
!	론리부정

C++에는 배타적론리합연산자가 없으므로 ||와 !연산자를 결합하여 실현한다.

2) 론리합

보물찾기놀이에서 사용자가 동쪽이나 서쪽으로 너무 멀리 가면 함정에 빠진다는것을 알려주는 프로그람을 론리합연산자(logical OR)를 리용하여 작성해보자.

(실례 3-22) 론리합연산자

```
#include <iostream>
using namespace std;
#include <process.h>
#include <conio.h>
int main()
{
  char dir = 'a';
  int x = 10, y = 10;
  cout << "중지하려면 Enter건을 누르시오.\n";
  while(dir != '\r')
  {
    cout << "\n동무의 위치는 " << x << "," << y;
    if(x < 5 || x > 15)
        cout << "\n주의! 여기에 함정이 있습니다.\n";
    cout << "\n방향건(북-n, 남-s, 동-e, 서-w)을 누르시오:";
    dir = getche();
    switch(dir)
    {
    case 'n': y--;
        break;
    case 's': y++;
        break;
    case 'e': x++;
        break;
    case 'w': x--;
        break;
    }
  }
  cout << endl;
  return 0;
}
```

식

$x < 5 || x > 15$

는 x가 5보다 작거나(선수가 서쪽으로 너무 멀리 갔을 때) x가 15보다 클 때(선수가 동쪽으로 너무 멀리 갔을 때) 참이다. 또한 ||연산자는 비교연산자 <와 >보다 우선순위가 낮으므로 이 식에는 괄호가 필요없다.

3) 론리부정

론리부정(logical NOT)연산자 !는 단항연산자(unary operator)이다. (대부분의 연산자는 2항연산자(binary operator)로서 두개의 연산수를 가진다. C++의 조건연산자는 유일한 3항연산자이다.) !의 동작은 연산수의 론리값을 반전하는것이다. 즉 !의 연산수가 참이면 결과는 거짓으로 되고 연산수가 거짓이면 참으로 된다.

실례로 (x == 7)은 x가 7과 같으면 참이지만 !(x == 7)은 x가 7과 같지 않으면 참이다. 이런 경우에는 !=연산자를 사용하여 x != 7라고 쓸수도 있다.

- 옹근수변수의 참과 거짓값

이제는 참과 거짓값을 가지는 식에 대한 표상을 가지게 되였다. 또한 론리식은 비교연산자를 포함한다. 그러나 사실상 하나의 옹근수식이 하나의 변수라고 하여도 참과 거짓값을 가진다. 식 x는 그것이 0이 아닌 옹근수이면 참이고 0이면 거짓이다. 이런 정황에서 !연산자를 적용하면 !x는 x가 0이면 참이고 0이 아닌 옹근수이면 거짓이다. 그것은 !이 x의 진리값을 반전하기때문이다.

보물찾기놀이에서 x와 y가 모두 7의 배수인 모든 위치에 버섯이 있다고 가정하자. x를 7로 나눈 나머지는 x % 7로 표시하며 x가 7에 의해 완전히 나누이여야 식의 값이 0이다. 버섯이 있는 위치를 지정하기 위하여서는

```
if( x % 7 == 0 && y % 7 == 0)
  cout << "여기에 버섯이 있습니다.\n" ;
```

그러나 비교연산자를 포함하지 않고도 !연산자를 사용하여 더 간단히 쓸수 있다.

```
if(!( x % 7) && !(y % 7))
```

이것은 우와 꼭같은 효과를 가진다.

론리연산자 &&와 ||는 비교연산자보다 우선순위가 낮다. 그러면 왜 x % 7과 y % 7에 괄호가 필요한가? 그것은 단항연산자 !가 비교연산자보다 우선순위가 높기때문이다.

4) 연산자의 우선순위

우선순위표에서 우에 있는 연산자의 우선순위는 아래에 있는 연산자보다 높다. 높은 우선순위를 가진 연산자는 낮은 우선순위를 가진 연산자보다 먼저 평가된다. 같은 행에 있는 연산자들은 우선순위가 같다. 괄호에 넣어서 식을 먼저 평가할수 있다. (표 3-3)

표 3-3. 연산자의 우선순위

연산자형	연산자
단항	!, ++, --, -
산수	곱하기 *, /, %
	더하기 +, -
비교	안같기 <, >, <=, >=
	같기 ==, !=

연산자형	연산자
론리	론리적 && 론리합 ||
조건	?:
대입	=, +=, -=, *=, /=, %=

제 4 절. 기타 조종명령문

C++에는 그밖에도 여러가지 조종명령문들이 있다. 이미 break는 switch명령문에서 보았다. 다른 명령문으로서 continue는 순환에서만 사용되며 goto는 다른곳으로 이행하게 한다.

1. break명령문

break명령문은 순환과 switch명령문으로부터 완료하게 한다. break가 실행된 다음의 명령문은 순환의 뒤에 오는 명령문이다. 그림 3-16은 break명령문의 조작을 보여준다.

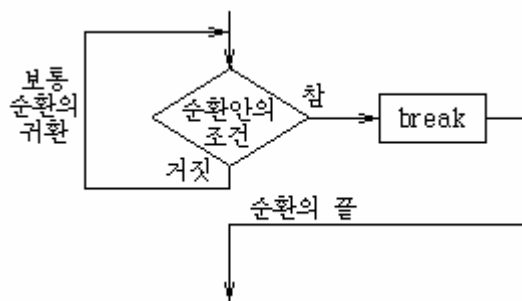



그림 3-16. break명령문의 조작

(실례 3-23) 씨수분포도

```
#include <iostream>
using namespace std;
#include <conio.h>
int main()
{
   const unsigned char WHITE = 'P';
   const unsigned char GRAY = ' ';
   unsigned char ch;
   for(int count=0; count<80*25-1; count++)
   {
      ch = WHITE;
      for(int j=2; j<count; j++)
          if(count % j == 0)
          {
             ch = GRAY;
```

```
                break;
            }
            cout << ch;
        }
        getch();
        return 0;
    }
```

실행결과 80×25행렬의 콘솔화면우의 매개 위치가 0～1999(즉 80×25－1)값으로서 표시된다. 어떤 위치에 있는 수값이 씨수이면 그 위치는 흰색으로 표시되고 그렇지 않으면 재색으로 표시된다.

안쪽의 for순환은 수값이 씨수가 아니라고 판단했을 때 문자 ch를 GRAY로 설정하고 안쪽 순환에서 벗어나기 위하여 break를 실행한다.

break는 항상 제일 안쪽 순환에서 벗어나게 한다. 이것은 서로 겹쌓인 구조에는 영향이 없다. 만일 순환안에 switch가 있다면 break는 순환밖으로 나오는것이 아니라 switch의 밖으로 나온다.

마지막 cout명령문은 도형문자를 출력하고 그다음 순환을 계속하여 다음의 수가 씨수인가를 검사한다.

- ASCII확장문자모임

이 프로그람에서는 확장 ASCII모임의 두개의 문자를 사용한다. 확장문자모임은 unsigned char형으로 표시한다. 값 219는 솔리드색블로크(흑백색표시체계에서는 흰색)를 의미하며 176은 재색을 의미한다.

실례 3-23에서는 마지막 행에서 getch()를 사용하여 프로그람이 끝날 때 화면이 스크롤되여 올라가지 않고 임의의 건을 누를 때까지 정지되게 한다. 또한 문자코드는 255까지인데 char형은 127까지이므로 문자변수로서 unsigned char형을 사용한다.

2. continue명령문

break명령문은 제일 안쪽 순환에서 나오게 한다. 그러나 때때로 기대하지 않던 결과가 생기여 순환의 꼭대기로 돌아가야 하는 경우가 있다. 이것은 continue로 실현한다. 정확히 말하면 continue는 순환본체의 닫긴 괄호까지 가서 꼭대기로 되돌아가게 한다. 그림 3-17은 continue의 조작을 보여준다.

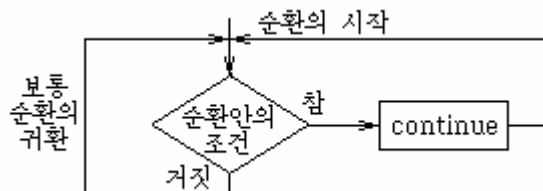



그림 3-17. continue명령문의 조작

여기에 실례 3-8을 변경한 실례 3-24가 있다. 실례 3-8은 나누기를 할 때 치명적인 결함을 가지고있다. 즉 사용자가 나누는 수로서 0을 입력하면 프로그람은 치명적인 오유를 일으키고 실행시 오유통보문 "Divide Error"를 발생시키고 완료한다. 실례 3-24에서는 이 상태를 구체적으로 취급한다.

(실례 3-24) continue명령문

```
#include <iostream>
using namespace std;
int main()
{
   long dividend, divisor;
   char ch;
   do
   {
      cout << "나누이는 수를 입력하십시오:";
      cin >> dividend;
      cout << "나누는 수를 입력하십시오:";
      cin >> divisor;
      if(divisor == 0)
      {
          cout << "나누는 수가 옳지 않습니다.\n";
          continue;
      }
      cout << "상=" << dividend / divisor;
      cout << ", 나머지=" << dividend % divisor;
      cout << "\n계속하겠습니까(y/n)?:";
      cin >> ch;
   } while(ch != 'n');
   return 0;
}
```

사용자가 나누는 수로서 0을 입력하면 프로그람은 오유통보문을 출력하고 continue에 의하여 순환의 웃끝으로 돌아온다. 실행결과는 다음과 같다.

```
나누이는 수를 입력하십시오: 10
나누는 수를 입력하십시오: 0
나누는 수가 옳지 않습니다.
나누이는 수를 입력하십시오:
```

이때 break명령문은 do순환을 완료하게 한다. 프로그람은 응답을 출력할 필요가 없다.

3. goto명령문

goto명령문을 사용하는것은 좋지 않다. 구조화프로그람작성원리를 알고있는 상태에서 goto명령문을 사용해보면 리해와 오유수정이 힘들다는것을 알수 있다.

goto를 사용하려면 코드안의 이행목적지에 표식을 삽입해야 한다. 표식은 항상 두점으로 끝난다. 예약어 goto뒤에 오는 표식이름은 그 표식에로 이행할수 있게 한다. 다음의 코드가 그것을 보여준다.

```
goto SystemCrash;
// 다른 명령문들
SystemCrash:
// goto에 의하여 조종은 여기로 넘어온다
```

요 약

비교연산자는 두개의 값이 서로 같은가, 큰가를 비교한다. 결과는 론리값으로서 true 또는 false이다. 거짓은 0으로, 참은 1 혹은 0아닌 값으로 표시된다.

C++에는 세개의 순환이 있다. for순환을 제일 많이 사용하며 순환의 실행회수를 알 때 사용한다. while순환과 do순환은 순환에서 순환완료조건이 요구될 때 사용된다. while순환은 한번도 실행되지 않을수 있으나 do순환은 항상 적어도 한번은 실행된다.

순환본체는 단순명령문 혹은 괄호에 넣은 복합명령문블로크일수 있다. 블로크안에서 정의된 변수는 블로크안에서만 볼수 있다.

네가지 종류의 분기명령문이 있다. if명령문은 검사식이 참이면 어떤 조작을 수행한다. if～else명령문은 검사식이 참이면 어떤 조작을 수행하고 참이 아니면 다른 조작을 수행한다. else if구조는 겹쌓인 if～else명령문을 리해하기 쉽게 쓰는 한가지 방법이다. switch명령문은 단일변수의 값에 따라서 코드의 여러절로 분기한다. 조건연산자는 검사식이 참일 때 어떤 값을, 거짓일 때는 다른 값을 돌려준다.

론리적과 론리합연산자는 두개의 론리식을 결합하여 다른 결과값을 내주며 론리부정연산자는 론리값을 참으로부터 거짓으로, 거짓으로부터 참으로 변경한다.

break명령문은 그것이 나타나는 제일 안쪽 순환이나 switch의 끝으로 조종을 보낸다. continue명령문은 그것이 위치하는 순환의 웃끝으로 조종을 옮긴다. goto명령문은 조종을 표식에로 보낸다.

우선순위는 어느 종류의 연산을 먼저 수행하는가를 지정한다. 우선순위는 단항, 산수, 비교, 론리, 조건, 대입연산자 등이다.

문 제

1. 비교연산자는 다음과 같이 조작한다.

① 한 연산수를 다른 연산수에 대입한다.

② 론리결과를 내준다.

③ 두개의 연산수를 비교한다.

④ 두개의 연산수를 론리적으로 결합한다.

어느것이 옳은가?

2. 비교연산자를 사용하여 먼저 var1이 var2과 같지 않으면 true를 보내는 식을 쓰시오.

3. -1은 참인가 거짓인가?

4. for명령문안에 사용하는 세개 식의 이름과 조작을 쓰시오.

5. 다중명령문순환본체를 가지는 for순환에서 반두점은 다음의 부분뒤에 있어야 한다.

① for명령문자체

② 다중명령문순환본체안의 닫긴 괄호

③ 순환본체안의 매개 명령문

④ 검사식

어느것이 옳은가?

6. for순환의 증분식은 순환변수를 감소시킬수 있는가?

7. 100～110의 수를 표시하는 for순환을 쓰시오.

8. 하나의 코드블로크를 어떻게 구분하는가?

9. 블로크안에서 정의된 변수는

① 프로그람안의 정의된 위치로부터 볼수 있다.

② 함수안의 정의된 위치로부터 볼수 있다.

③ 블로크안의 정의된 위치로부터 볼수 있다.

④ 함수전체에서 볼수 있다.

어느것이 옳은가?

10. 100～110의 수를 while순환을 사용하여 표시하는 명령문을 쓰시오.

11. 비교연산자는 산수연산자보다 우선순위가 높은가?

12. do순환에서 순환본체는 몇번 실행되는가?

13. 100～110의 수를 출력하는 do순환을 쓰시오.

14. 변수 age가 21보다 크면 Yes를 출력하는 if명령문을 쓰시오.

15. 서고함수 exit()는

① 그것이 들어있는 순환으로부터 완료하게 한다.

② 그것이 들어있는 블로크로부터 완료하게 한다.

③ 그것이 들어있는 함수로부터 완료하게 한다.

④ 그것이 들어있는 프로그람으로부터 완료하게 한다.

어느것이 옳은가?

16. if～else명령문을 사용하여 변수 age가 21보다 크면 Yes를 출력하고 아니면 No를 출력하시오.

17. getche()서고함수는

① 임의의 건을 눌렀을 때 그 문자를 돌려준다.

② Enter건을 눌렀을 때 그 문자를 돌려준다.

③ 임의의 건을 눌렀을 때 그 문자를 화면에 표시한다.

④ 그 문자를 화면에 표시하지 않는다.

어느것이 옳은가?

18. 사용자가 Enter건을 누를 때 cin으로부터 얻어지는 문자는 무엇인가?

19. else는 어느 if와 쌍을 이루는가?

20. else if구조는 겹쌓인 if～else구조를 재구성하여 얻어진다. 옳은가?

21. switch를 사용하여 변수 ch가 'y'이면 Yes, 'n'이면 No를 출력하고 그렇지 않으면 Unknown response를 출력하시오.

22. 조건연산자를 사용하여 speed가 55보다 크면 ticket를 1로, 그렇지 않으면 0으로 설정하는 명령문을 쓰시오.

23. &&와 ||연산자는

① 두개의 수값을 비교한다.

② 두개의 수값을 결합한다.

③ 두개의 론리값을 비교한다.

④ 두개의 론리값을 결합한다.

어느것이 옳은가?

24. limit가 55이고 speed가 55보다 크다는 론리연산자를 포함하는 식을 쓰시오.

25. 다음 종류의 연산자 즉 론리, 단항, 산수, 대입, 비교, 조건연산자들을 우선순위에 따라 배렬하시오.

26. break명령문은

① 제일 안쪽에 있는 순환으로부터만 완료하게 한다.

② 제일 안쪽에 있는 switch로부터만 완료하게 한다.

③ 모든 순환과 switch로부터 완료하게 한다.

④ 제일 안쪽에 있는 순환이나 switch로부터 완료하게 한다.

어느것이 옳은가?

27. 순환안에서 continue를 실행하면 조종은 어디로 가는가?

28. goto명령문은 조종이

① 연산자에로 가게 한다.

② 표식에로 가게 한다.

③ 변수에로 가게 한다.

④ 함수에로 가게 한다. 어느것이 옳은가?

련습문제

1. 어떤 수의 곱하기표를 만드는 프로그람을 작성하시오. 사용자가 수를 입력하면 프로그람은 다음과 같이 10행 10렬로 된 표를 출력한다.

```
수를 입력하시오: 7
    7   14   21   28   35   42   49   56   63   70
   77   84   91   98  105  112  119  126  133  140
  147  154  161  168  175  182  189  196  203  210
      ...
```

2. 사용자가 float형의 온도를 입력하면 그것이 화씨온도인 경우에 섭씨온도로, 섭씨온도인 경우에 화씨온도로 변환하는 프로그람을 작성하시오. 프로그람과의 대화는 다음과 같다.

```
화씨온도를 섭씨온도로 변환하려면 1을 누르시오.
섭씨온도를 화씨온도로 변환하려면 2를 누르시오: 1
화씨온도를 입력하시오: 70
섭씨온도는 21.111111
```

3. 건반입력을 받아들이는 >>와 같은 연산자는 수자들의 렬을 수로 변환할수 있어야 한다. 이러한 기능을 수행하는 프로그람을 작성하시오. 사용자가 6자리까지 수자를 입력한 다음 결과값을 long형의 옹근수로 표시하시오. 수자들은 getchar()를 사용하여 한 문자씩 개별적으로 읽어야 한다. 현재의 수에 10을 곱하고 그다음 새 수자를 더하는 방법으로 수를 만드시오.(ASCII코드의 수값문자를 수로 변환하려면 수값문자에서 '0' 또는 48을 던다.) 프로그람과의 대화는 다음과 같다.

```
수를 입력하시오: 123456
수는 : 123456
```

4. 4기능전자수산기를 만드시오. 사용자는 수, 연산자, 수를 입력한다.(류동소수점수가능) 그다음 산수연산 즉 두수의 더하기, 덜기, 곱하기, 나누기를 실행한다. 연산을 선택할 때 switch명령문을 사용하시오. 마지막에 결과를 표시한다. 계산을 끝마칠 때 프로그람은 사용자에게 계산을 계속하겠는가를 물어야 한다. 여기에 'y' 또는 'n'으로 응답할수 있다. 프로그람과의 대화는 다음과 같다.

```
첫 연산수, 연산자, 둘째 연산수를 입력하시오: 10/3
답 = 3.333333
계산을 계속하겠습니까(y/n)? y
첫 연산수, 연산자, 둘째 연산수를 입력하시오: 12+100
답 = 112
계산을 계속하겠습니까(y/n)? n
```

5. 화면에 x로 이루어진 바른 3각형을 표시하는 프로그람을 작성하시오. 그 형식은 다음과 같다.

```
       X
      XXX
     XXXXX
    XXXXXXX
   XXXXXXXXX
```

3각형의 높이는 20이여야 한다. 겹쌓인 두개의 순환을 사용하시오.

6. 사용자가 수값을 입력하면 그 수의 차례곱을 계산하고 0을 입력하면 계산을 끝내는 프로그람을 작성하시오. while과 do순환을 사용해보시오.

7. 저금소에 보통저금형식으로 저금해둔 돈이 얼마나 되는가를 계산하는 프로그람을 작성하시오. 사용자는 초기저금액, 저금기간(년), 퍼센트로 된 리자률을 입력해야 한다. 프로그람과의 대화는 다음과 같다.

초기저금액을 입력하시오: 3000
저금기간(년)을 입력하시오: 10
리자률(%률)을 입력하시오: 5.5
현재 저금액수는 5124.43원입니다.

8. 사용자가 두개 리의 토지면적을 정보, 평 단위로 각각 입력하면 그것을 더하여 정보, 평으로 표시하는 프로그람을 작성하시오. 프로그람과의 대화는 다음과 같다.

첫째 리의 토지면적을 입력하시오: 100 1500
둘째 리의 토지면적을 입력하시오: 120 1700
토지의 총면적은 221정보 200평입니다.
계산을 계속하겠습니까(y/n)? n

9. 6명의 손님에 대한 식사를 자리가 4석인 방에서 준비하였다고 하자. 이때 6명중 4명은 자리에 앉고 나머지 2명은 서야 한다. 6명의 손님중 4명은 몇가지 방법으로 의자에 배렬할수 있는가? 4개자리에 대한 6명 손님의 가능한 배렬수는 6×5×4×3=360이다. 임의의 인원수와 임의의 자리수에 대하여 가능한 배렬수를 계산하는 프로그람을 작성하시오. 손님이 좌석수보다 많다고 가정하시오. for순환을 사용하시오.

10. 련습 7과 반대인 프로그람을 작성하시오. 최종저금액수대신에 그것이 몇년동안 저금된 결과인가를 계산하여야 한다.

11. 3기능전자수산기를 만드시오. 전자수산기는 정보, 평으로 된 토지의 량을 더하고 덜고 표준단위(㎢와 ㎡)로 변환할수 있어야 한다.

12. 분수용 4기능전자수산기를 만드시오. 두개의 분수에 대하여 다음과 같이 연산한다.

더하기: a/b + c/d = (a*d + b*c) / (b*d)
덜기: a/b - c/d = (a*d - b*c) / (b*d)
곱하기: a/b * c/d = (a*c) / (b*d)
나누기: a/b / c/d = (a*d) / (b*c)

사용자가 첫째 연산수, 연산자, 둘째 연산수를 입력하면 프로그람은 그 결과를 표시하고 계산을 계속하겠는가를 물어야 한다.

제 4 장. 구조체와 렬거

이미 float, char, int와 같은 기본자료형에 대하여 보았다. 이러한 형의 변수들은 정보의 한개 항목 즉 높이, 량과 같은것을 표시한다. 그러나 어떤 점의 자리표는 x와 y값으로 이루어지고 종업원과 같은 자료는 문자렬로 된 단어들로 구성되므로 여러개의 단순변수들로 복잡한 실체들을 조작해야 한다.

이것을 C++로 실현하려면 구조체를 사용하여야 한다.

이 장의 첫 부분에서는 구조체의 선언과 정의, 구조체성원의 호출, 겹쌓인 구조체, 객체와 자료형으로서의 구조체에 대하여 설명하고 둘째 부분에서는 렬거에 대하여 설명한다.

제 1 절. 구조체

구조체(structure)는 단순변수들의 집합이다. 구조체안의 변수들은 서로 다른 형일수 있다. 즉 일부는 int, 다른것은 float형일수 있다. 이것은 배렬과 다르다. 배렬에서는 모든 원소들의 형이 같아야 한다. 구조체안의 자료항목을 구조체의 성원(member)이라고 한다.

C프로그람작성에서는 보통 구조체를 고급한 기능으로 간주하고있다. 그러나 C++프로그람작성자에게 있어서 구조체는 객체와 클라스를 리해하는데서 매우 중요하다. 사실상 구조체의 문법은 클라스의 문법과 거의 비슷하다. 구조체는 자료의 집합이며 클라스는 자료와 함수의 집합이다. 따라서 구조체를 알면 클라스와 객체에 대하여 쉽게 리해할수 있다. C++와 C의 구조체는 Pascal과 같은 언어에서 레코드로 취급된다.

두개의 옹근수와 한개의 류동소수점수를 포함하는 구조체변수를 고찰하자. 이 구조체는 기계의 부분품목록에서 한개의 항목을 표시한다. 부분품은 형번호와 부분품번호, 단가를 성원으로 가진다.

실례 4-1에서는 구조체 Part를 선언하고 part1이라는 구조체변수를 정의한다. 그 다음 그 변수의 성원들에 값을 대입하고 값들을 표시한다.

(실례 4-1) 부분품명세(구조체사용)

```
#include <iostream>
using namespace std;
struct Part
{
   int modelNumber;
   int partNumber;
   float cost;
};
int main()
```

```
  {
    Part part1;
    part1.modelNumber = 6244;
    part1.partNumber = 373;
    part1.cost = 217.55F;
    cout << "형번호=" << part1.modelNumber;
    cout << ", 부분품번호=" << part1.partNumber;
    cout << ", 단가=" << part1.cost << "원\n";
    return 0;
  }
```

이 프로그람의 실행결과는 다음과 같다.

```
형번호= 6244, 부분품번호= 373, 단가= 217.55원
```

실례 4-1은 세가지 주요한 개념 즉 구조체선언, 구조체변수의 정의, 구조체성원의 호출에 대하여 보여주었다.

1. 구조체선언

구조체선언은 구조체를 어떻게 구성하는가 즉 구조체가 어떤 성원들을 가지는가를 지정한다. 실례에서 구조체선언은 다음과 같다.

```
struct Part
{
  int modelNumber;
  int partNumber;
  float cost;
};
```

- 구조체선언의 문법

예약어 struct는 구조체선언을 받아들인다. struct뒤에는 구조체이름(struct name, tag)이 온다.(실례에서 Part) 구조체성원들의 선언(즉 modelNumber, partNumber, cost)은 괄호안에 들어있다. 닫긴 괄호뒤에 반두점이 오며 그로써 구조체선언이 끝난다. 구조체에서 반두점의 사용은 코드블로크에서와 다르다. 순환에서 사용되는 코드블로크와 선언, 함수들은 괄호에 의하여 구분되지만 최종괄호뒤에는 반두점을 사용하지 않는다. 그림 4-1은 구조체선언의 문법을 보여준다.

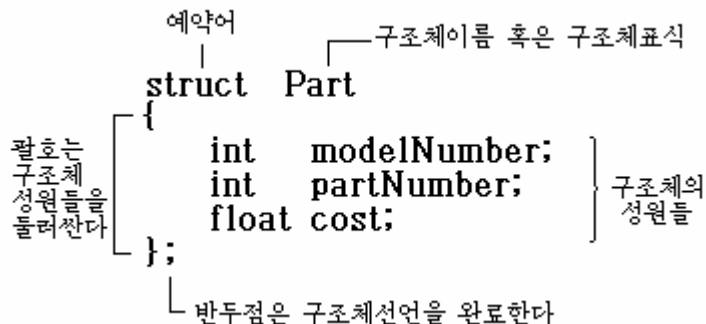



그림 4-1. 구조체선언의 문법

- 구조체선언의 사용

구조체선언은 Part형의 변수를 참조하기 위한 자료형으로서 사용된다. 구조체선언은 한개의 변수를 정의하는것이 아니다. 구조체선언은 기억기안에서 림시적인 공간을 할당하지 않으며 어떤 림시변수를 명명하지도 않는다. 구조체선언은 바로 그러한 형의 구조체변수들이 정의될 때 기억공간에 어떻게 만들어지는가를 지정한다. 이것을 그림 4-2에서 보여준다.

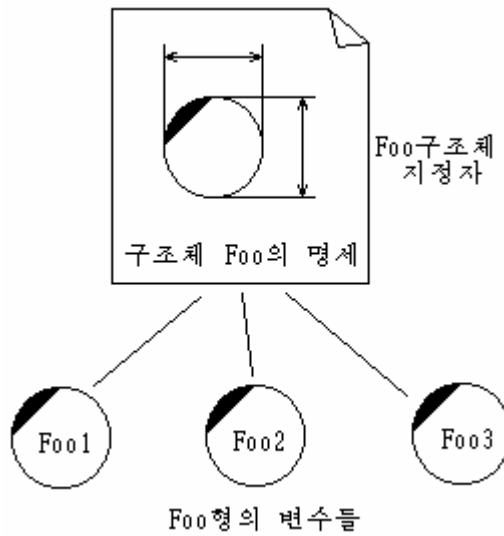



그림 4-2. 구조체와 구조체변수

2. 구조체변수의 정의

main()의 첫 명령문

```
Part part1;
```

은 Part구조체형의 part1이라는 변수를 정의한다. 이 정의는 기억기에 part1용의 공간을 예약한다. 즉 part1의 모든 성원들 다시말하여 modelNumber, partNumber, cost를 충분히 보관할수 있는 공간을 예약한다. int형의 두개 변수에 각각 4byte씩, float형의 변수에 4byte를 할당한다. 그림 4-3은 part1이 기억기에 어떻게 배치되는가를 보여준다.(그림은 2byte옹근수를 보여준다.)

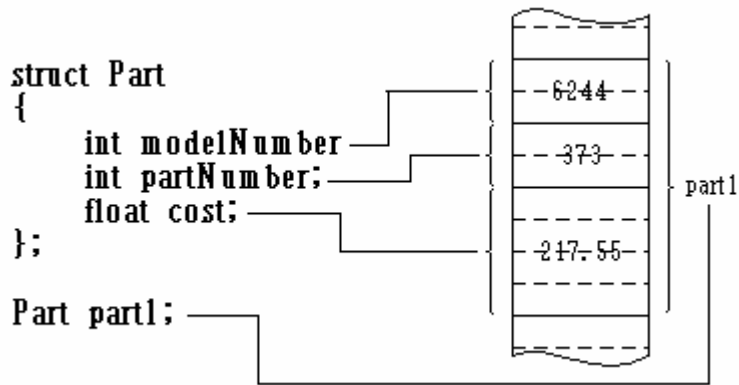



그림 4-3. 기억기에서 구조체성원들의 배치

Part구조체를 새 자료형의 명세로서 생각할수 있다. 그러나 구조체변수를 정의하는 형식은 int와 같은 기본자료형을 정의할 때와 같다.

```
Part patr1;
int var1;
```

이러한 류사성은 우연한것이 아니다. C++의 하나의 목적은 사용자정의자료형의 문법과 조작을 될수록 기본자료형과 비슷하게 만드는데 있다. C에서는 구조체변수정의에 예약어 struct를 포함하여야 한다. 즉

```
struct Part part1;
```

그러나 C++에서는 예약어가 필요없다.

3. 구조체성원호출

일단 구조체변수를 정의한 다음 점연산자(dot operator)에 의하여 그 성원을 호출할수 있다. 실례로 첫째 성원에 다음과 같이 값을 줄수 있다.

```
part1.modelNumber = 6244;
```

구조체성원은 세개의 요소 즉 구조체변수의 이름(part1), 점연산자(.) 그리고 성원이름(례를 들면 modelNumber)으로 쓸수 있다. 이것은 《 part1의 성원 modelNumber》를 의미한다. 점연산자의 실제이름은 성원호출연산자(member access operator)이지만 보통 긴 이름을 사용하지 않는다.

점연산자를 포함하는 식의 첫째 요소는 구조체의 이름(Part)이 아니라 특정한 구조체변수(이 경우에 part1)의 이름이다. 변수의 이름은 그림 4-4에 보여주는것처럼 변수가 한개이상 있을 때(실례로 part1, part2) 그것들을 구별하는데 사용된다.

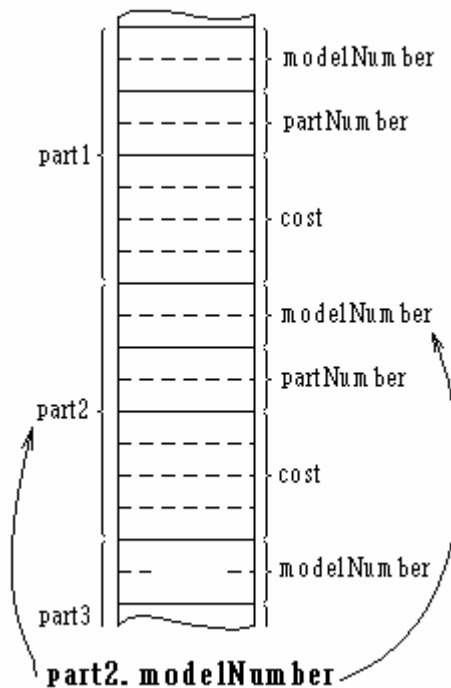



그림 4-4. 점연산자

구조체성원은 마치도 다른 변수처럼 취급된다. 명령문

```
part1.modelNumber = 6244;
```

에서 이 성원에는 표준대입연산자(normal assignment operator)에 의하여 값 6244가 할당된다. 또한 실례에서는 다음과 같이 cout명령문을 통하여 성원들을 출력한다.

```
cout << "형번호=" << part1.modelNumber;
```

이 명령문은 구조체성원의 값을 출력한다.

4. 구조체의 기타 특성

구조체문법과 사용법의 일부 추가적특성들을 고찰하자.

- 선언과 정의의 결합

실례 4-1에서는 구조체선언과 정의를 하는데 두개의 명령문을 리용하였다. 그러나 실례 4-2에서처럼 한개 명령문으로 할수 있다.

(실례 4-2) 부분품명세(이름없는 구조체사용)

```
#include <iostream>
using namespace std;
struct              // 이름이 없다
{
   int modelNumber;
   int partNumber;
   float cost;
} part1;
int main()
```

```
{
    part1.modelNumber = 6244;
    part1.partNumber = 373;
    part1.cost = 217.55F;
    cout << "형번호=" << part1.modelNumber;
    cout << ", 부분품번호=" << part1.partNumber;
    cout << ", 단가=" << part1.cost << "원\n";
    return 0;
}
```

실례에서는 다음과 같이 구조체정의를 위한 명령문을 따로 쓰지 않는다.

```
Part part1;
```

그대신 변수이름 part1은 선언의 뒤에 붙어있다.

```
struct              // 이름이 없다
{
    int modelNumber;
    int partNumber;
    float cost;
} part1;
```

구조체형의 변수를 프로그람의 다른 부분에서 더는 정의할 필요가 없으면 우와 같이 구조체선언에서 구조체이름을 삭제할수 있다.

구조체선언과 정의를 결합하는 이 방법은 여러행의 프로그람을 함께 쓰는 가장 단순한 방법이다. 일반적으로 이 방법은 선언과 정의를 따로따로 사용하는 경우보다 명백하지 않고 유연성이 적다.

- 구조체성원의 초기화

실례 4-3은 구조체변수를 정의할 때 구조체성원들을 초기화하는 방법을 보여준다. 또한 주어진 구조체형의 변수를 한개이상 가질수 있다는것을 보여준다.

(실례 4-3) 구조체변수의 초기화

```
#include <iostream>
using namespace std;
struct Part
{
    int modelNumber;
    int partNumber;
    float cost;
};
int main()
{
    Part part1 = { 6244, 373, 217.55F };
    Part part2;
    cout << "형번호=" << part1.modelNumber;
    cout << ", 부분품번호=" << part1.partNumber;
    cout << ", 단가=" << part1.cost << "원\n";
    part2 = part1;
    cout << "형번호=" << part2.modelNumber;
```

```
    cout << ", 부분품번호=" << part2.partNumber;
    cout << ", 단가=" << part2.cost << "원\n";
    return 0;
}
```

이 실례는 Part형의 두개 변수 part1과 patr2를 정의한다. 그리고 part1을 초기화하고 그 성원값들을 출력하며 part1을 part2에 대입하고 part2의 성원들을 출력한다.

그 실행결과는 다음과 같다.

```
형번호= 6244, 부분품번호= 373, 단가= 217.55원
```

part1구조체변수의 성원들은 그 변수가 정의될 때 초기화된다. 즉

```
Part part1 = { 6244, 373, 217.55F };
```

구조체성원들에 대입하려는 값들은 반점으로 구분하여 대괄호안에 넣는다. 목록안의 첫째 값은 첫째 성원에, 둘째 값은 둘째 성원에 각각 대입된다.

- 대입명령문에서 구조체변수

실례 4-3에서 알수 있는것처럼 하나의 구조체변수를 다른 구조체변수에 대입할수 있다. 즉

```
part2 = part1;
```

part1의 매 성원들의 값은 part2의 대응하는 성원들에 대입된다. 큰 구조체인 경우에는 많은 성원을 가질수 있으므로 이러한 대입명령문은 콤퓨터가 상당한 량의 작업을 하게 한다.

같은 형의 구조체변수들에 대해서만 어떤 구조체변수를 다른 구조체변수에 대입할 수 있다.

어떤 구조체형의 변수를 다른 구조체형의 변수에 대입하려고 하면 번역프로그람은 오유를 통보한다.

- 측정실례

구조체를 각이한 종류의 정보를 보관하는데 사용하는 방법을 고찰하자.

건물설계프로그람을 만들려고 한다. 건물의 크기는 두 단위 m와 cm로 보관하는것이 좋다. 실례 4-4에서는 이것을 구조체에 의하여 실현한다. 이 프로그람은 Distance형의 두개의 치수를 모두 더하는 방법을 보여준다.

(실례 4-4) 거리표시에 구조체사용

```
#include <iostream>
using namespace std;
struct Distance
{
    int meters;
    float centies;
};
int main()
{
    Distance d1, d3;
    Distance d2 = { 11, 6.25 };
```

```
    cout << "\n메터를 입력하시오:";
    cin >> d1.meters;
    cout << "센치메터를 입력하시오:";
    cin >> d1.centies;
    d3.centies = d1.centies + d2.centies;
    d3.meters = 0;
    if(d3.centies >= 100.0)
    {
       d3.centies -= 100.0;
       d3.meters++;
    }
    d3.meters += d1.meters + d2.meters;
    cout << d1.meters << "m " << d1.centies << "cm + ";
    cout << d2.meters << "m " << d2.centies << "cm = ";
    cout << d3.meters << "m " << d3.centies << "cm\n";
    return 0;
}
```

여기서 구조체 Distance에는 두개의 성원 meters와 centies가 있다. centies변수는 소수부를 가지므로 float형으로 하고 meters는 항상 옹근수이므로 int형으로 한다.

건물의 길이 d1와 d3을 초기화하지 않고 정의하고 d2은 11m 6.25cm로 초기화한다. 프로그람은 사용자에게 m와 cm로 된 길이를 입력할것을 요구하며 길이 d1와 d2을 더하여 총 길이 d3을 얻는다. 끝으로 프로그람은 두개의 길이와 새로 계산한 총 길이를 표시한다.

```
메터를 입력하시오: 10
센치메터를 입력하시오: 6.75
10m 6.75cm + 11m 6.25cm = 22m 1cm
```

d3 = d1 + d2과 같은 프로그람명령문에 의하여 두 길이를 더할수 없다. 그것은 Distance형의 변수들을 더하는 방법을 알고있는 루틴이 C++에 없기때문이다. +연산자는 float와 같은 기본형에서는 리용되지만 사용자가 정의한 Distance에 대해서는 리용할수 없다. 그러나 클라스를 사용하면 사용자정의자료형에 대해서도 더하기를 할수 있다.

5. 겹쌓인 구조체

구조체안에 다른 구조체를 넣을수 있다. Distance형의 두개 변수 length와 width를 사용하여 방의 크기 즉 길이와 너비를 보관하는 자료구조체를 정의할수 있다. 즉

```
struct Room
{
   Distance length;
   Distance width;
};
```

실례 4-5는 방의 크기를 표시하는데 Room구조체를 사용한다.

(실례 4-5) 겹쌓인 구조체

```
#include <iostream>
using namespace std;
struct Distance
{
   int meters;
   float centies;
};
struct Room
{
   Distance length;
   Distance width;
};
int main()
{
   Room dinning;
   dinning.length.meters = 13;
   dinning.length.centies = 6.5;
   dinning.width.meters = 10;
   dinning.width.centies = 0.0;
   float m = dinning.length.meters + dinning.length.centies / 100;
   float w = dinning.width.meters + dinning.width.centies / 100;
   cout << "방의 면적은 " << m * w << "㎡\n";
   return 0;
}
```

이 프로그람은 Room형의 한개 변수 dinning을 다음의 행에서 정의한다.

```
Room dinning;
```

그다음 이 구조체의 성원들에 값을 대입한다.

- 겹쌓인 구조체성원의 호출

한 구조체가 다른 구조체안에 놓여있으면 구조체성원들을 호출할 때 점연산자를 두번 적용해야 한다.

```
dinning.length.meters = 13;
```

이 명령문에서 dinning은 구조체변수의 이름이고 length는 바깥쪽 구조체(Room)의 성원이름, meters는 안쪽 구조체(Distance)의 성원이름이다. 이 명령문은 변수 dinning의 length성원의 meters성원을 호출하고 거기에 값 13을 대입한다.

그림 4-5는 그 과정을 보여준다.

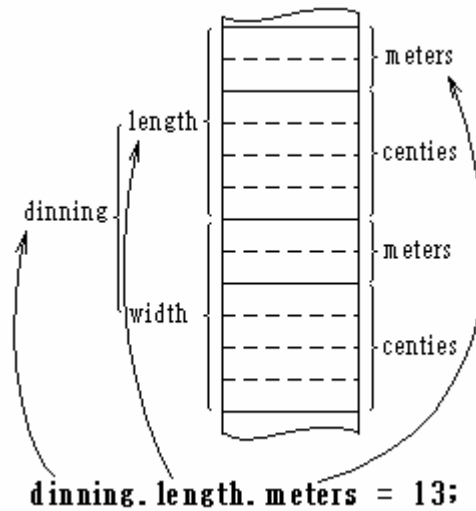



그림 4-5. 점연산자와 겹쌓인 구조체

일단 값이 dinning의 성원들에 대입된 다음에 프로그람은 그림 4-6에서 보여준것처럼 그 방의 면적을 계산한다.

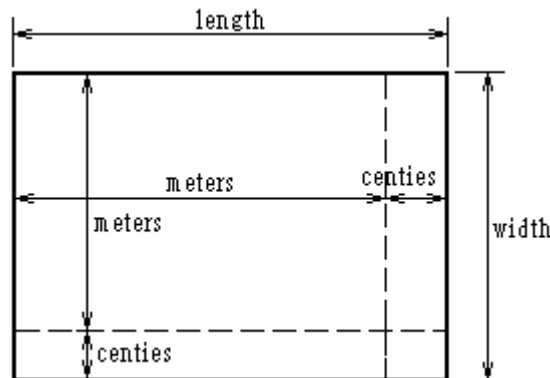



그림 4-6. meters와 centies로 구성된 면적

프로그람은 면적을 계산하기 위하여 Distance형의 변수 length와 width를 m로 거리를 표시하는 float형변수 m과 w로 변환한다. m과 w의 값은 Distance의 meters성원과 centies성원을 100으로 나눈 값을 더하여 얻는다.

meters성원은 더하기 전에 자동적으로 float형으로 변환되므로 결과는 float형이다. 그다음 l과 w변수를 곱하여 면적을 계산한다.

- 사용자정의형변환

프로그람에서는 Distance형의 두개 길이를 float형의 두개 길이 즉 변수 m과 w로 변환한다. 또한 방의 면적을 ㎡로 변환하여 면적을 보관하는 변수에 하나의 류동소수점수로 보관한다. 실행결과는 다음과 같다.

방의 면적은 136.5㎡

어떤 형의 값을 다른 형의 값으로 변환하는 방법은 사용자정의자료형을 사용하는 프로그람들에서 많이 사용한다.

- 겹쌓인 구조체의 초기화

그러면 구조체변수자체에 구조체들을 포함하는 경우 변수를 어떻게 초기화하는가?

다음의 명령문은 변수 dinning을 실례 4-5의 프로그람에서 주어진 값들로 초기화한다.

```
Room dinning = {{13, 6.5}, {10, 0.0}};
```

Room안에 들어있는 Distance형의 매개 구조체는 제각기 초기화된다. 이때 값들을 반점으로 구분하여 괄호안에 넣는다.

첫째 Distance는 {13, 6.5}로 초기화되고 둘째는 {10, 0.0}로 초기화된다. 그다음 Room변수를 두개의 거리값들로 초기화하기 위하여 그것들을 다시 반점으로 구분하여 괄호안에 넣는다.

- 겹쌓임의 깊이

리론적으로는 구조체를 임의의 깊이까지 겹쌓을수 있다. 아빠트를 설계하는 프로그람에서 그러한 명령문을 찾을수 있다.

```
appartment1.lanmdryRoom.washingMachine.width.meters
```

- 카드놀이 실례

몇가지 실례를 고찰하자. 카드놀이를 모의할 때 구조체를 사용할수 있다. 여기서 한 사람이 세개의 카드를 보여주고 그다음 그것들을 책상우에 뒤집어놓고 카드들을 여러번 바꾸어놓는다. 어떤 특정한 카드가 어데 있는가를 정확히 알아맞추면 그 사람이 이긴다. 카드를 빠른 속도로 바꾸어놓으면 상대편은 카드들의 위치를 잊어버린다.

여기에 카드를 표시하는 구조체가 있다.

```
struct Card
{
   int number;
   int suit;
};
```

이 구조체는 카드의 한조(suit)의 번호를 보관하는데 성원들을 사용한다. number는 2～14, 여기서 11,12,13,14는 각각 J(jack), Q(queen), K(king), A(ace)를 표시한다. suit는 0～3으로 표시되고 여기서 네개의 수값은 각각 클라브(club), 다이야몬드(diamond), 하트(heart), 스페이드(spade)를 의미한다.

(실례 4-6) 구조체를 사용한 카드놀이

```
#include <iostream>
using namespace std;
const int clubs        = 0;
const int diamonds     = 1;
const int hearts       = 2;
const int spades       = 3;
```

```
const int jack =      11;
const int green =        12;
const int king =      13;
const int ace =       14;
struct Card { int number; int suit; };
int main()
{
  Card temp, chosen, prize;
  int position;
  Card card1 = { 7, clubs };
  cout << "card1는 클라브 7입니다.\n";
  Card card2 = { jack, hearts };
  cout << "card2는 하트 쟈크(J)입니다.\n";
  Card card3 = { ace, spades};
  cout << "card3는 스페이드 에이스(A)입니다.\n"; prize = card3;
  cout << "card1과 card3을 교체합니다.\n";
  temp = card3;
  card3 = card1;
  card1 = temp;
  cout << "card2와 card3을 교체합니다.\n";
  temp = card3;
  card3 = card2;
  card2 = temp;
  cout << "card1과 card2를 교체합니다.\n";
  temp = card2;
  card2 = card1;
  card1 = temp;
  cout << "스페이드 에이스(A)의 위치(1,2,혹은 3)가 어디입니까? ";
  cin >> position;
  switch(position)
  {
  case 1: chosen = card1; break;
  case 2: chosen = card2; break;
  case 3: chosen = card3; break;
  }
  if(chosen.number == prize.number && chosen.suit == prize.suit)
     cout << "옳습니다!\n";
  else
     cout << "틀렸습니다.\n";
  return 0;
}
```

프로그람의 실행결과는 다음과 같다.

```
card1은 클라브 7입니다.
card2는 하트 쟈크(J)입니다.
card3은 스페이드 에이스(A)입니다.
card1과 card3을 교체합니다.
card2와 card3을 교체합니다.
card1과 card2를 교체합니다.
```

```
스페이드 에이스(A)의 위치(1,2,혹은 3)가 어디입니까? 3
틀렸습니다.
```

이 실례에서는 카드를 잘못 선택한다.(정확한 답은 2이다.)

프로그람은 카드번호와 조번호값으로서 const int형의 변수들을 정의하는것으로부터 시작한다. 그다음 Card구조체를 지정하고 세개의 초기화하지 않은 Card형변수 temp, chosen, prize를 정의한다. 또한 세개의 카드 card1, card2, card3을 정의하고 세개의 임의의 카드값들로 초기화한다. 그리고 사용자의 정보에 대응하여 이 카드들의 값을 출력한다. 그다음 카드변수 prize를 이 카드값들의 하나로서 설정하고 그것을 기억한다. 이 카드는 선수가 경기마감에 알아맞추려고 하는 위치의 카드이다.

다음으로 프로그람은 카드를 재배치한다. 첫째와 넷째, 둘째와 셋째, 첫째와 둘째 카드들을 각각 교체한다. 그리고 사용자에게 질문한다.

끝으로 프로그람은 특정카드가 어느 위치에 있는가를 묻는다. 프로그람은 카드변수 chosen을 이 위치의 카드로 설정하고 chosen과 prize를 비교한다. 일치하면 그 선수가 이기고 그렇지 않으면 틀린 경우이다.

그러면 어떻게 하면 카드를 간단히 교체하겠는가?

```
temp = card3;
card3 = card2;
card2 = temp;
```

카드가 구조체를 표시한다하더라도 그것들을 아주 자연스럽게 이동할수 있고 구조체와 작업할 때는 대입연산자 =의 능력이면 충분하다.

구조체인 경우에는 서로 비교할수 없다. 즉

```
if(chosen == prize)
```

라고 할수 없다. 왜냐하면 Card구조체에 대하여 알고있는 ==연산자용으로 작성된 루틴이 없기때문이다. 그러나 연산자를 재정의하면 해결할수 있다.

6. 구조체와 클라스

구조체는 보통 자료만 보관하는데 사용한다. 클라스는 자료와 함수를 둘다 보관하는데 사용한다. 그러나 사실상 C++에서는 구조체에 자료와 함수를 모두 보관할수 있다.(C에서는 자료만 보관할수 있다.) C++에서 구조체와 클라스사이의 문법적차이는 적으며 리론적으로 거의 같이 사용할수 있다. 그러나 대다수 C++프로그람작성자들은 구조체를 자료만 보관하는데 사용하고 클라스는 자료와 함수를 둘다 보관하는데 사용한다.

제 2 절. 렬거

구조체는 사용자정의자료형을 제공하는 방법의 하나이다. 자체의 자료형을 정의하는 다른 방법은 렬거이다. C++의 이 기능은 구조체보다 중요하지 않다. 렬거에 대한

지식이 없어도 아주 좋은 객체지향프로그람을 쓸수 있다. 그렇지만 렬거는 C++프로그람에서 많이 사용되며 자체의 자료형을 정의함으로써 프로그람작성을 단순화하고 명백하게 한다.

1. 렬거형의 선언

렬거형(enumerated type)은 옹근수형의 상수값들의 목록을 알고있을 때 정의한다. 실례 4-7에서는 요일에 렬거형을 사용한다.

(실례 4-7) 렬거형

```
#include <iostream>
using namespace std;
enum DaysOfWeek { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
int main()
{
   DaysOfWeek day1, day2;
   day1 = Mon;
   day2 = Thu;
   int diff = day2 - day1;
   cout << "day2과 day1의 날자 차이는 " << diff << endl;
   if(day1 < day2)
      cout << "day1은 day2의 전이다\n";
   return 0;
}
```

렬거선언은 그 형에서 허용되는 모든 이름들의 모임을 정의한다. 이때 가능한 값들을 렬거자(enumerator)라고 한다. 렬거형DaysOfWeek는 7개의 렬거자 즉 Sun, Mon, Tue, Wed, Thu, Fri, Sat를 가진다. 그림 4-7은 enum선언의 문법을 보여준다.

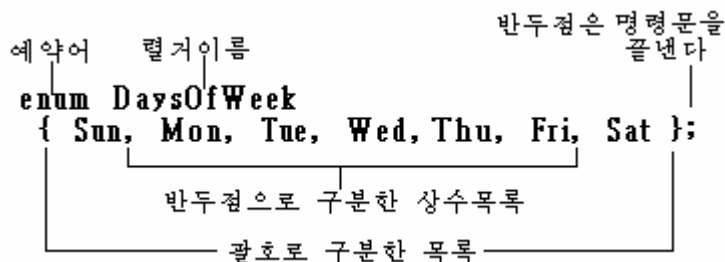



그림 4-7. enum지정자의 문법

렬거는 가능한 모든 값들의 목록이다. 이것은 int의 지정과 다르다. 례를 들면 값들의 범위도 주어진다. enum에서는 매개의 가능한 값에 개별적으로 이름을 주어야 한다. 그림 4-8은 int와 enum의 차이를 보여준다.

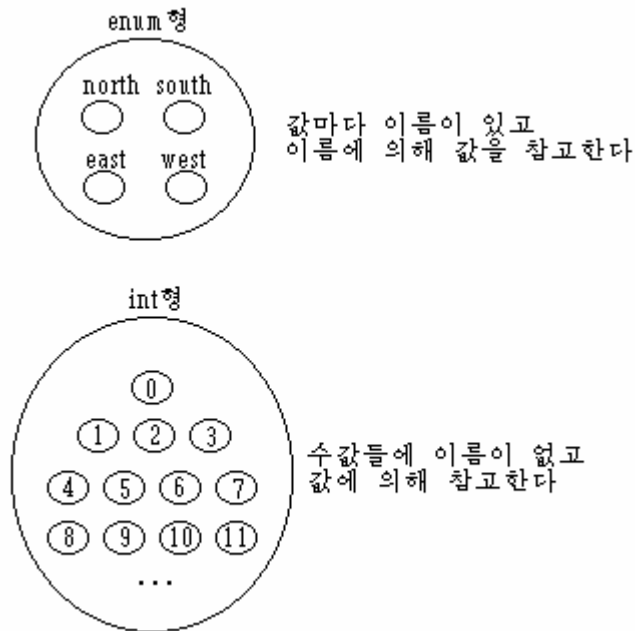



그림 4-8. int와 enum의 사용법

enum형 DaysOfWeek를 일단 선언한 다음에는 이 형의 변수를 정의할수 있다. 이 실례에서는 이 형의 변수 day1과 day2을 정의하였다. 즉

```
DaysOfWeek day1, day2;
```

C에서는 형이름앞에 예약어 enum을 써야 한다. 즉

```
enum DaysOfWeek day1, day2;
```

그러나 C++에서는 enum이 필요없다.

day1, day2과 같은 렬거형변수들은 enum선언에서 렬거한 임의의 값을 가질수 있다. 실례에서는 그 변수들에 Mon과 Thu를 주었다. 선언에서 렬거하지 않은 값은 쓸수 없다. 그러한 명령문으로서

```
day1 = halloween;
```

은 틀린다.

enum형에 대하여 표준산수연산자를 사용할수 있다. 프로그램에서는 두개의 수를 던다. 또한 비교연산자를 사용할수 있다. 프로그람의 실행결과는 다음과 같다.

```
day2과 day1의 날자차이는 2
day1은 day2의 전이다
```

산수 및 비교연산자의 사용은 일부 enum형에서 잘 어울리지 않는다. 실례로 다음과 같이

```
enum Pets { cat, dog, hamster, canary, ocelot };
```

라고 선언한다면 dog + canary 혹은 cat < hamster와 같은 식이 무엇을 의미하는지 명백히 알수 없다.

렬거는 내부적으로 옹근수로 취급된다. 따라서 렬거에 대한 산수 및 비교연산이

가능해진다. 보통 목록안의 첫 이름에는 값 0이 주어지고 다음 이름에는 값 1, …이 주어진다. 실례 4-7에서 Sun~Sat는 옹근수값 0~6을 가진다.

enum형에 대한 산수연산은 옹근수값에 대하여 수행되므로 번역프로그람은 enum 변수들이 사실상 옹근수라는것을 알고있어도 작성자는 렬거형이 자기의 우월성을 발휘하도록 하여야 한다.

```
day1 = 5;
```

라고 하면 번역프로그람은 경고를 내보낸다. enum이 실제로 옹근수라는것을 잊어버리는것이 좋다.

2. 렬거형과 론리형

다음의 실례는 사용자가 입력한 문장의 단어수를 계수한다. 실례 3-13과 달리 단어수를 계산하기 위해 공백을 계수하지 않는다. 그대신 그림 4-9와 같이 비공백문자들의 문자렬이 공백으로 바뀌는 위치를 계수한다.

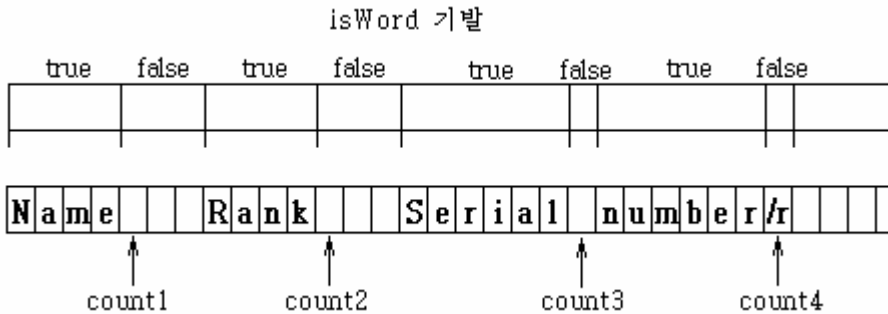



그림 4-9. 실례 4-8의 동작

이 방법은 단어들사이에 여러개의 공백이 입력되면 정확한 단어개수를 얻지 못한다.(여전히 타브와 기타 공백문자는 조정할수 없다.) 실례 4-8은 두개의 렬거자만 가진 렬거를 보여준다.

(실례 4-8) 렬거형, 문장안의 단어수 계산

```
#include <iostream>
using namespace std;
#include <conio.h>
enum ItsAWord { No, Yes };
int main()
{
  ItsAWord isWord = No;
  char ch = 'a';
  int wordCount = 0;
  cout << "문장을 입력하십시오:\n";
  do
  {
     ch = getche();
     if(ch == ' ' || ch == '\r')
```

```
        {
            if(isWord == Yes)
            {
                wordCount++;
                isWord = No;
            }
        }
        else
            if(isWord == No)
                isWord = Yes;
    } while(ch != '\r');
    cout << "\n단어수는 " << wordCount << endl;
    return 0;
}
```

do순환에서 프로그람의 주기는 건반으로부터의 한개문자읽기이다. 프로그람은 공백을 찾을 때까지 비공백문자를 뛰여넘는다. 공백을 찾았을 때 프로그람은 단어를 계수한다. 그다음 문자를 찾을 때까지 공백을 뛰여넘고 다시 공백을 찾을 때까지 문자를 계수한다. 이 작업은 프로그람에게 현재 단어안에 있는가, 공백문자렬안에 있는가 하는것을 기억할것을 요구한다. 프로그람은 ItsAWord형의 enum변수 isWord를 기억한다. 이 형은 명령문

```
enum ItsAWord { No, Yes };
```

에서 지정된다.

ItsAWord형의 변수들은 두개의 가능한 값 No와 Yes를 가진다. 이 목록이 No로 시작되므로 변수값은 false를 가리키는 값 0으로 주어진다.(이때 bool형의 변수를 사용할수도 있다.)

isWord변수는 프로그람이 시작할 때 No로 설정된다. 프로그람이 처음으로 비공백문자와 만나면 단어안에 있다는것을 가리키도록 Yes로 설정한다. 프로그람은 다음 공백을 찾을 때까지 이 값을 유지하며 공백이 발견되면 No로 다시 설정된다. 사실 No는 값 0, Yes는 값 1을 가지지만 이 수값들을 사용하지 않는다. if(isWord == Yes)대신에 if (isWord)를 사용하고 if(isWord == No)대신에 if(!isWord)를 사용할수도 있다. 이것은 그리 좋은 방법이 아니다.

또한 프로그람에서 둘째 if명령문둘레에 여분의 괄호를 요구하므로 else는 첫 if와 쌍을 이룬다.

- 카드놀이의 조직화

렬거형의 마지막 실례를 고찰하자. 실례 4-6에서 카드 한조를 표시하기 위해 const int형의 상수들을 정의했다.

```
const int clubs     = 0;
const int diamonds  = 1;
const int hearts    = 2;
const int spades    = 3;
```

이러한 목록은 부자연스럽다. 렬거형을 사용하도록 실례 4-6을 수정한것이 실례 4-9이다.

(실례 4-9) 구조체를 사용한 카드놀이

```
#include <iostream>
using namespace std;
const int jack =  11;
const int green = 12;
const int king =  13;
const int ace =   14;
enum Suit { clubs, diamonds, hearts, spades };
struct Card { int number; Suit suit; };
int main()
{
   Card temp, chosen, prize;
   int position;
   Card card1 = { 7, clubs };
   cout << "card1는 클라브 7입니다.\n";
   Card card2 = { jack, hearts };
   cout << "card2는 하트 쟈크(J)입니다.\n";
   Card card3 = { ace, spades};
   cout << "card3는 스페이드 에이스(A)입니다.\n"; prize = card3;
   cout << "card1과 card3을 교체합니다.\n";
   temp = card3;
   card3 = card1;
   card1 = temp;
   cout << "card2와 card3을 교체합니다.\n";
   temp = card3;
   card3 = card2;
   card2 = temp;
   cout << "card1과 card2를 교체합니다.\n";
   temp = card2;
   card2 = card1;
   card1 = temp;
   cout << "스페이드 에이스(A)의 위치(1,2,혹은 3)가 어디입니까? ";
   cin >> position;
   switch(position)
   {
   case 1: chosen = card1; break;
   case 2: chosen = card2; break;
   case 3: chosen = card3; break;
   }
   if(chosen.number == prize.number && chosen.suit == prize.suit)
      cout << "옳습니다!\n";
   else
      cout << "틀렸습니다.\n";
   return 0;
}
```

여기에서는 실례 4-6에서 사용한 Suit의 정의부분을 enum렬거로 바꾸었다.

```
enum Suit { clubs, diamonds, hearts, spades };
```

이것은 const변수의 사용보다 더 명백한 방법이다. Suit의 가능한 값들을 더 정확히 알수 있으며 다른 값을 대입하려고 하면 례를 들어

```
card1.suit = 5;
```

는 번역프로그람으로부터 경고를 일으킨다.

3. 옹근수값의 지정

렬거선언에서 첫 렬거자는 옹근수값 0, 둘째는 1,…으로 된다. 이 순서는 =기호를 사용하여 0이 아닌 출발값을 지정하여 변경할수 있다. 실례로 Suit를 0대신 1로 시작하게 하려면

```
enum Suit { clubs=1, diamonds, hearts, spades };
```

라고 쓴다.

뒤에 오는 이름들은 이 값으로 시작되는 값들로 설정된다. 즉 diamonds는 2, hearts는 3, spades는 4이다. 같기기호를 사용하여 렬거자에 특정한 값을 줄수 있다.

4. 렬거형의 입출력

C++입출력명령문들은 enum형의 렬거자들을 인식하지 못한다. 실례로 다음의 코드를 실행할수 있겠는가?

```
enum Direction { North, South, East, West };
Direction dir1 = South;
cout << dir1;
```

일반적으로 South라고 출력된다고 생각할수 있다. 그러나 C/C++는 enum형변수를 옹근수로 취급하므로 출력은 1이다.

- 다른 실례

여기에 렬거자료선언의 실례가 있다.

```
enum Months { Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
enum Switch { Off, On };
enum Meridian { am, pm };
```

요 약

이 장에서는 구조체와 렬거에 대하여 설명하였다. 구조체는 C++의 중요한 요소이다. 구조체는 클라스와 같은 문법을 가진다. 사실상 클라스는 함수를 포함하는 구조체이다.

보통 구조체는 여러 자료항목들을 모두 묶어서 하나의 실례를 구성하는데 사용된다.

구조체선언은 구조체를 이루는 변수들을 선언한다. 정의는 구조체변수용으로 기억기를 할당한다.

구조체변수는 어떤 환경에서는 개별적인 단위로 취급되지만 다른 환경에서는 점연산자를 사용하여 그 성원들을 개별적으로 호출한다.

렬거는 사용자정의자료형으로서 고정값들의 목록으로 제한된다. 선언은 자료형을 주며 가능한 값들을 지정한다. 이 값들을 렬거자라고 한다. 정의는 구조체형의 변수를 창조한다. 번역프로그람은 내부적으로 렬거변수를 옹근수로 취급한다.

구조체를 렬거형과 혼돈하지 말아야 한다. 구조체는 자료의 여러가지 집합을 하나의 실체로 묶어놓는 강력하고 융통성있는 방법이다. 렬거는 형을 선언할 때 렬거값들로 이루어지는 고정모임을 취급할수 있는 변수정의를 가능하게 한다.

문 제

1. 구조체는

① 같은 자료형의 묶음이다.

② 련관된 자료항목들의 묶음이다.

③ 사용자정의이름을 가지는 옹근수들의 묶음이다.

④ 변수들의 묶음이다.

어느것이 옳은가?

2. 구조체와 클라스의 문법은 같은가?

3. 구조체선언의 닫긴괄호뒤에는 어떤 기호가 놓이는가?

4. 세개의 변수 즉 int형의 hrs, mins, secs를 가지는 Time이라는 구조체를 쓰시오.

5. 구조체선언은 기억기에 변수용공간을 창조하는가?

6. 구조체성원을 호출할 때 점연산자의 왼변에 있는 식별자는

① 구조체성원의 이름이다.

② 구조체이름이다.

③ 구조체변수의 이름이다.

④ 예약어 struct의 이름이다.

어느것이 옳은가?

7. Time구조체변수 time2의 hrs성원을 11로 설정하는 명령문을 쓰시오.

8. struct Time형의 세개의 변수가 정의되여있고 이 구조체가 세개의 int형성원을 가진다면 구조체변수들은 각각 몇byte씩 차지하는가?

9. struct Time형의 변수 time1의 성원들을 hrs=1, mins=10, secs=59로 초기화하는 정의를 쓰시오.

10. 한 구조체변수를 같은 형의 다른 구조체변수에 대입할수 있는가?

11. 변수 temp를 fido변수의 dogs성원의 paw성원으로 설정하는 명령문을 쓰시오.

12. 렬거는

① 서로 다른 자료형의 항목들

② 련관된 자료변수들

③ 사용자정의이름을 가지는 상수들

④ 상수값들

의 묶음이다. 어느것이 옳은가?

13. B1, B2, SS, B3, RF, CF, LF, P, C를 값으로 가지는 Players라는 렬거를 선언하는 명령문을 쓰시오.

14. 렬거형Players를 문제 13과 같이 선언하였다고 가정할 때 변수 kim과 ri를 정의하고 그것들에 각각 LF와 P값을 대입하시오.

15. 문제 13,14의 명령문을 가정하였을 때 다음 명령문이 옳은가를 말하시오.

① Kim = B;

② Ri = SS;

③ LF = ri;

④ Difference = kim - ri;

16. enum형의 처음 세개의 렬거자들은 보통 어떤 값을 가지는가?

17. 렬거자 fast, middle, slow를 가지는 Speeds라는 렬거를 선언하는 명령문을 쓰시오. 그리고 이 세개의 이름에 옹근수값 78, 45, 33을 주시오.

18. enum IsWord { NO, YES };가 enum IsWord { YES, NO }보다 좋은 리유를 말하시오.

련습문제

1. 212-767-8900과 같은 전화번호는 3개 부분 즉 지역코드(212), 구역코드(767), 번호(8900)로 되여있다. 구조체를 사용하여 전화번호의 3개 부분을 따로따로 보관하는 프로그람을 작성하시오. 우선 구조체 Phone형을 창조하고 Phone형의 구조체변수 두개를 창조하시오. 하나는 초기화하고 다른 하나는 사용자가 전화번호를 입력하게 하고 두개의 전화번호를 화면에 표시하시오. 프로그람과의 대화는 다음과 같다.

```
나의 전화번호는 212-767-8900
동무의 전화번호는 415-515-1202
```

2. 2차원평면우의 한점은 두개의 수 x와 y자리표로 표시된다. 례를 들면 (4, 5)는 수평축의 오른쪽으로 4단위, 수직축의 우로 5단위를 표시한다. x자리표가 두 점의 x자리표들의 합, y자리표는 개개의 y자리표들의 합으로 되는 새 점을 정의할수 있다. 점을 표시할 때 Point라고 부르는 구조체를 사용하는 프로그람을 작성하시오. 세개의

점을 정의하고 사용자가 그중 두개의 점을 입력하시오. 그다음 두개 점들의 합으로서 셋째 점을 설정하고 그 점의 자리표를 출력하시오. 프로그람과의 대화는 다음과 같다.

P1의 자리표를 입력하시오: 3 4
P2의 자리표를 입력하시오: 5 7
P1+P2의 자리표 = 8, 11

3. 방의 부피를 모의하기 위하여 Distance형변수 3개를 사용하는 Volume이라는 구조체를 창조하시오. Volume형의 변수를 어떤 값으로 초기화하고 그 방의 부피를 계산하여 결과를 출력하시오. 부피를 계산하기 위하여 Distance변수로부터 매개 치수를 m로 환산한 float형변수로 변환하고 그 값들을 곱하시오.

4. 두개의 성원 즉 종업원번호(int형), 종업원의 생활비(원, float형)를 보관하는 구조체 Employee를 창조하시오. 사용자가 3명의 종업원자료를 입력하고 그것들을 Employee구조체형의 3개 변수에 보관한 다음 매 종업원의 정보를 표시하는 프로그람을 작성하시오.

5. 3개의 성원 즉 년, 월, 일(모두 int형)을 표시하는 구조체 Date를 창조하시오. 사용자가 2003/8/15형식으로 날자를 입력하면 그것을 Date형변수에 보관하고 변수로부터 값을 얻어서 같은 형식으로 출력하는 프로그람을 작성하시오.

6. C++입출력명령문들은 렬거형을 자동적으로 인식하지 못한다. 이 제한을 극복하기 위하여 switch명령문을 사용하여 렬거변수를 표시하는 방법으로 렬거변수의 실제값들로 변환할수 있다. 실례로 종업원의 형을 가리키는 값들을 가진 렬거형을 고찰하자. 즉

enum EType { Laborer, Secretary, Manager, Accountant, Executive, Researcher };

사용자가 첫문자 'l', 's', 'm', 'a', 'e', 'r'를 입력하여 형을 지정하고 선택된 형을 enum EType형변수의 값으로 보관하고 이 형의 옹근단어를 표시하는 프로그람을 작성하시오. 이 대화실례는 다음과 같다.

렬거형(첫 문만)을 입력하시오.(Laborer, Secretary, Manager, Accountant, Executive, Researcher): a
렬거형은 Accountant입니다.

이때 두개의 switch명령문이 요구된다. 하나는 입력용, 다른 하나는 출력용이다.

7. enum EType형의 변수와 구조체 Date형의 변수를 련습 4의 Employee구조체에 추가하시오. 사용자가 3명의 종업원 매 사람에 대하여 4가지 정보 즉 종업원번호, 생활비, 직종, 입직날자를 입력하는 프로그람을 작성하시오. 프로그람은 Employee형의 3개 변수에 이 정보를 보관하고 내용을 표시한다.

8. 분수구조체 Fraction을 창조하시오. Fraction의 2개 성원은 나누이는 수와 나누는 수(둘다 int형)이다. 분수를 구조체 Fraction형의 변수에 보관하는 프로그람을 작성하시오.

9. int형의 3개 성원 시, 분, 초를 가지는 Time이라는 구조체를 창조하시오. 시간

값을 시, 분, 초로 입력하는 프로그람을 작성하시오. 이것은 12:59:59형식일수도 있고 매개 수를 개별적으로 입력할수도 있다. 그다음 Time형의 변수에 시간을 보관하고 이 시간값을 초단위로 환산하여 출력하시오. 총 시간은

```
long totalSects = t1.hours * 3600 + t1.minutes * 60 + t1.seconds;
```

10. 정보와 평으로 된 토지면적을 보관하는 GAera구조체를 창조하시오. 성원으로서 정보는 int, 평은 float형이다. 사용자가 ㎡로 된 면적(double)을 입력하면 그것을 정보, 평 단위로 변환하여 GAera형구조체변수에 보관하고 정보, 평 형식으로 표시하도록 하시오.

11. 련습 9의 Time구조체를 사용하여 사용자로부터 12:59:59형식의 두개의 시간값을 얻어서 Time구조체변수에 보관하고 매개를 초(int)로 변환하고 더하여 결과를 다시 시, 분, 초형식으로 변환하여 Time구조체에 보관한 다음 12:59:59형식으로 표시하시오.

12. 3장의 4기능분수수산기프로그람을 수정하시오. 련습 8의 Fraction형변수에 분수를 보관하시오.

제 5 장. 함수

함수는 여러개의 프로그람명령문을 하나의 단위로 묶은것이며 거기에 이름을 준다. 이 단위를 프로그람의 다른 부분에서 호출할수 있다.

함수를 사용하는 가장 중요한 리유는 프로그람의 개념적인 조직화에 있다. 한개 프로그람을 함수들로 나누는것은 구조화프로그람작성에서 중요한 원칙의 하나이다. 객체지향프로그람작성에서는 프로그람을 조직화하는 더 강력한 방법을 제공해준다.

함수를 사용하는 또 하나의 리유는 프로그람의 크기를 줄이는데 있다. 프로그람에서 한번이상 나타나는 명령들의 렬은 한개의 함수로 만들기 위한 후보로 된다. 함수코드는 기억기의 유일한 장소에 보관된다. 이것은 프로그람이 전과정에 함수를 여러번 실행하는 경우에도 마찬가지이다. 그림 5-1은 프로그람의 다른 부분들에서 함수를 호출하는 방법을 보여준다.

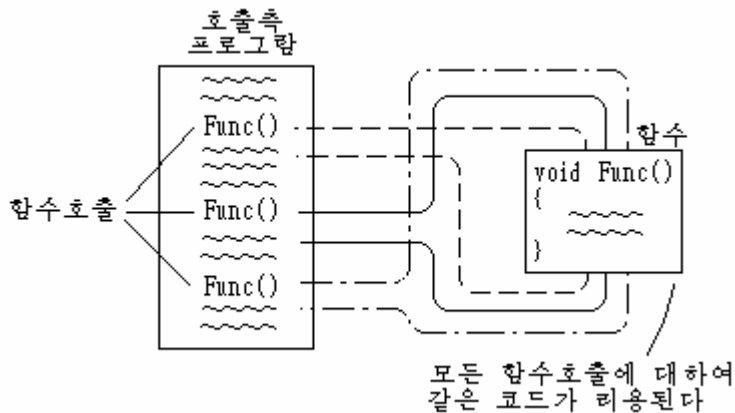



그림 5-1. 함수의 조종흐름

C/C++의 함수는 다른 언어들에서 보조루틴(subroutine) 혹은 수속(procedure)과 비슷하다.

이 장에서는 함수의 정의와 선언, 인수와 돌림값, 참고인수, 재정의된 함수, 기정인수, 변수의 기억등급에 대하여 설명한다.

제 1 절. 간단한 함수

실례 5-1은 한행에 45개의 별표를 출력하는 간단한 함수를 보여준다. 실례프로그람은 표를 하나 만들며 표읽기를 쉽게 하기 위하여 별표행들을 삽입한다.

(실례 5-1) 간단한 함수

```
#include <iostream>
using namespace std;
void StarLine();
```

```
int main()
{
  StarLine();
  cout << "자료형의 범위" << endl;
  StarLine();
  cout << "char  -128 ~ 127" << endl;
  cout << "short -32,768 ~ 32,767" << endl;
  cout << "int    체계의존" << endl;
  cout << "long  -2,147,483,648 ~ 2,147,483,647" << endl;
  StarLine();
  return 0;
}
void StarLine()
{
  for(int j=0; j<45; j++)
     cout << '*';
  cout << endl;
}
```

프로그람의 출력은 다음과 같다.

```
*********************************************
자료형의 범위
*********************************************
char  -128 ~ 127
short -32,768 ~ 32,767
int    체계의존
long  -2,147,483,648 ~ 2,147,483,647
*********************************************
```

프로그람은 두개의 함수 main()과 StarLine()으로 이루어진다. main()이 있는 프로그람은 이미 많이 보았다. 프로그람에 함수를 추가할 때 세가지 요소 즉 함수선언, 함수에로의 호출, 함수정의가 요구된다.

1. 함수선언

번역프로그람에 변수를 소개하지 않고서는 사용할수 없는것처럼 함수도 소개하지 않고서는 사용할수 없다. 함수선언방법에는 두가지가 있다. 제일 많이 사용하는 방법은 함수를 호출하기 전에 선언하는것이다. 다른 방법은 함수를 호출하기 전에 정의하는것이다. 실례 5-1에서는 함수 StarLine()을 다음과 같이 선언한다.

```
void StarLine();
```

선언은 번역프로그람에게 앞으로 StarLine이라는 함수를 사용한다는것을 알린다. 예약어 void는 함수의 돌림값이 없다는것을 지정하고 빈괄호는 인수를 가지지 않는다는것을 지정한다. 함수가 인수를 가지지 않는다는것을 지적할 때 괄호안에 void를 쓸수도 있다. C에서는 자주 이렇게 썼으나 C++에서는 쓰지 않아도 된다.

함수선언 그자체는 완전한 명령문이므로 반두점으로 끝난다. 또한 함수선언은 그

함수에 대한 모형을 제공해주므로 원형선언(prototype)이라고 한다.

함수선언은 번역프로그람에게 《선언이후의 프로그람에서 이 함수를 사용하므로 함수자체를 보기전에 그에 대한 선언을 보시오.》라고 통보한다.

2. 함수호출

main()으로부터 함수가 세번 호출된다. 함수호출은 다음과 같다.

```
StarLine();
```

함수호출에 필요한것은 함수이름과 그 뒤의 괄호이다. 함수호출문법은 선언과 비슷하지만 돌림값형을 쓰지 않는다. 함수호출은 반두점으로 끝난다. 호출명령문을 실행하면 함수가 실행된다. 즉 조종은 함수에로 넘어가고 함수정의안의 명령문들이 실행되며 그다음 조종은 함수호출뒤의 명령문으로 돌아온다.

3. 함수정의

끝으로 함수정의를 고찰하자. 함수정의는 함수의 실제코드를 포함한다. 여기에 StarLine()의 정의가 있다.

```
void StarLine()
{
   for(int j=0; j<45; j++)
      cout << '*';
   cout << endl;
}
```

함수정의는 선언자(declarator)라고 부르는 행과 그 뒤에 오는 함수본체로 이루어진다. 함수본체는 함수를 이루는 명령문들을 포함하며 대괄호에 의해 구분된다.

선언자는 선언과 일치해야 한다. 즉 같은 함수이름을 사용하고 인수들이 놓이는 순서와 형이 같아야 하며 같은 형의 돌림값을 가져야 한다.

선언자는 반두점으로 끝나지 않는다. 그림 5-2는 함수선언, 함수호출, 함수정의의 문법을 보여준다.

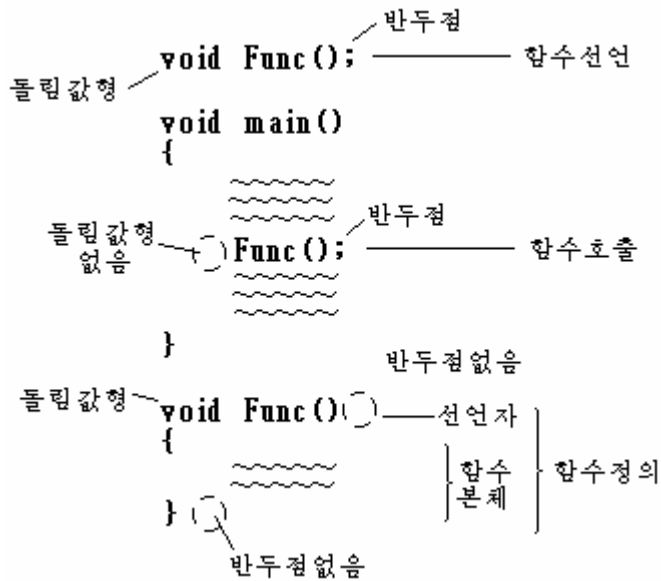



그림 5-2. 함수의 문법

함수를 호출할 때 조종은 함수본체의 첫 명령문에로 넘어간다. 그다음 함수본체안의 다음 명령문들이 실행되고 닫긴 괄호가 나타나면 조종은 호출한 프로그람에로 돌아온다.

표 5-1. 함수의 요소

요소	목적	실례
선언	함수이름, 인수형, 돌림값을 지정한다. 번역프로그람은 그 함수가 후에 나타난다는것을 통보한다.	void Func();
호출	함수를 실행하게 한다.	Func();

정의	함수 그자체이다. 대괄호안에 코드행들을 포함한다.	void Func()
		{
		// 코드행
		}

선언자	정의의 첫 행	void Func()

4. 서고함수와의 비교

이미 앞에서 서고함수를 사용하여 보았지만 프로그람의 코드로서 미리 준비되여있는 서고함수에 대한 호출이 가능하다. 실례로

```
ch = getche();
```

에서 서고함수에 대한 선언과 정의는 어디에 있는가?

그 선언은 프로그람의 선두에서 지적한 머리부파일(getche()는 CONIO.H)에 있다. 실행코드로 번역된 정의는 프로그람을 구축할 때 프로그람에 자동적으로 결합되는 서고함수안에 있다.

서고함수를 사용할 때에는 그 선언이나 정의를 쓸 필요가 없지만 자체의 함수를 쓸 때에는 원천파일에 선언과 정의를 써야 한다.

5. 선언의 생략

프로그람에 함수를 삽입하는 둘째 방법은 함수선언을 생략하고 그 함수를 처음으로 호출하기 전에 함수정의를 하는것이다.

실례 5-2는 실례 5-1을 변경한것이다.

(실례 5-2) 함수호출전에 함수정의

```
#include <iostream>
using namespace std;
void StarLine()
{
   for(int j=0; j<45; j++)
      cout << '*';
   cout << endl;
}
int main()
{
   StarLine();
   cout << "자료형의 범위" << endl;
   StarLine();
   cout << "char  -128 ~ 127" << endl;
   cout << "short -32,768 ~ 32,767" << endl;
   cout << "int    체계의존" << endl;
   cout << "long  -2,147,483,648 ~ 2,147,483,647" << endl;
   StarLine();
   return 0;
}
```

이 방법은 간단하지만 선언을 생략하였으므로 융통성이 적다. 이 방법을 사용하는 경우 함수가 많으면 작성자는 함수를 호출하기 전에 선언하도록 함수들을 배렬하는데 상당한 품을 들여야 한다. 때로는 이것이 불가능할수 있다. 또한 많은 작성자들은 실행이 시작되는 main()을 프로그람의 처음에 두는것을 더 좋아한다. 그러므로 보통 먼저 선언하고 main()으로부터 프로그람을 시작하는 첫째 방법을 선택한다.

제 2 절. 함수에 인수넘기기

인수는 프로그람으로부터 함수에로 넘기는 자료(실례로 int값)의 한부분이다. 인수는 함수가 여러값에 대하여 조작하게 하거나 지어는 함수를 호출하고 프로그람의 요구에 따라서 서로 다른 일을 하게 한다.

1. 상수넘기기

실례 5-2의 StarLine()은 항상 별표를 45개 출력한다. 그러나 실례 5-3에서는 임의의 문자를 임의의 개수로 출력한다. 여기서는 출력하려는 문자와 그 출력회수를 넘기는데 인수를 사용한다.

(실례 5-3) 함수인수

```
#include <iostream>
using namespace std;
void RepChar(char, int);
int main()
{
   RepChar('/', 43);
   cout << "자료형의 범위" << endl;
   RepChar('=', 23);
   cout << "char  -128 ~ 127" << endl;
   cout << "short -32,768 ~ 32,767" << endl;
   cout << "int    체계의존" << endl;
   cout << "long  -2,147,483,648 ~ 2,147,483,647" << endl;
   RepChar('/', 43);
   return 0;
}
void RepChar(char ch, int n)
{
   for(int j=0; j<n; j++)
      cout << ch;
   cout << endl;
}
```

새 함수는 RepChar()이다. 함수선언은 다음과 같다.

```
void RepChar(char, int);
```

괄호안의 항목들은 RepChar()에 보내는 인수들의 자료형 즉 char와 int이다.

함수를 호출할 때에는 특정값(이 경우에 상수)들이 괄호안의 해당한 위치에 삽입된다.

```
RepChar('/', 43);
```

이 명령문은 RepChar()에게 43개의 사선으로 된 행을 출력할것을 명령한다. 호출할 때 제공되는 값들은 선언에서 지정된 형이여야 한다. 즉 첫 인수 '/'문자는 char형, 둘째 인수값 43은 int형이여야 한다. 선언과 정의안의 행들도 역시 일치하여야 한다.

다음번의 RepChar()의 호출

```
RepChar('=', 23);
```

은 이 함수에 23개의 =기호를 한행에 출력할것을 명령한다. 세번째 호출 역시 43개의 사선을 출력한다. 결과는 다음과 같다.

```
///////////////////////////////////////////
자료형의 범위
=======================
```

```
char  -128 ~ 127
short -32,768 ~ 32,767
int    체계의존
long  -2,147,483,648 ~ 2,147,483,647
////////////////////////////////////////////
```

호출측프로그람은 함수에 '/'와 43과 같은 인수를 제공한다. 함수안에서 인수값을 보관하는데 사용되는 변수를 파라메터라고 한다. 즉 RepChar()에서 파라메터는 ch와 n이다. 함수정의안의 선언자는 파라메터들의 자료형과 이름을 둘다 지정한다.

```
void RepChar(char ch, int n)  // 선언자는 파라메터이름과 자료형을 지정한다
```

함수안에서 파라메터이름 ch와 n은 마치도 표준변수와 같이 사용된다. 선언자안에서 파라메터들의 배치는

```
char ch;
int n;
```

과 같은 명령문들로 그것들을 정의한것과 같다.

함수를 호출할 때 파라메터는 자동적으로 호출측프로그람이 넘기는 값들로 초기화된다.

2. 변수넘기기

실례 5-3에서 인수는 상수 '/'과 43 같은것들이다. 인수로서 상수대신 변수를 넘기는 실례를 고찰하자. 실례 5-4는 실례 5-3의 RepChar()와 같지만 반복하는 문자와 회수를 사용자가 지적한다.

(실례 5-4) 변수인수

```
#include <iostream>
using namespace std;
void RepChar(char, int);
int main()
{
   char chin;
   int nin;
   cout << "문자를 입력하시오: ";
   cin >> chin;
   cout << "반복회수를 입력하시오: ";
   cin >> nin;
   RepChar(chin, nin);
   return 0;
}
void RepChar(char ch, int n)
{
   for(int j=0; j<n; j++)
      cout << ch;
   cout << endl;
}
```

이 프로그람과의 대화는 다음과 같다.

문자를 입력하시오: +
반복회수를 입력하시오: 20
++++++++++++++++++++

여기서 main()의 chin과 nin은 RepChar()에 대한 인수로서 사용된다.

```
RepChar(chin, nin);
```

인수로서 사용된 변수들의 자료형은 함수선언과 정의에서 지정된 형들과 일치하여야 한다. 즉 chin은 char, nin은 int형이여야 한다.

3. 값에 의한 넘기기

실례 5-4에서 함수호출의 chin과 nin이 가지는 특정값들은 함수에로 넘어온다. 상수를 함수에로 넘길 때처럼 함수는 이 변수(인수)들의 값을 보관하기 위하여 새 변수(파라메터)들을 창조한다. 함수는 새 변수들에 선언에서 주어진 이름과 자료형(즉 char형의 ch와 int형의 n)을 준다. 함수는 넘어온 값들로 파라메터들을 초기화하고 파라메터들은 함수본체안의 명령문들에 의하여 다른 변수들처럼 호출된다.

함수가 자기에게 넘어온 인수의 사본을 창조하는 방법으로 인수를 넘기는것을 값에 의한 넘기기(passing by value)라고 한다.

그림 5-3은 함수에 인수를 값에 의해 넘길 때 함수안에서 새 변수들이 창조되는 과정을 보여준다.

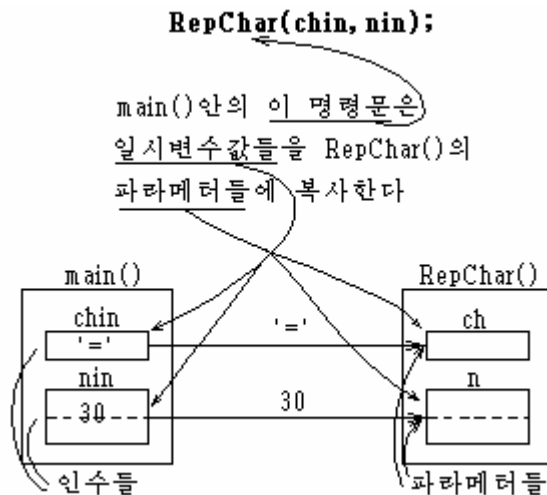



그림 5-3. 값에 의한 넘기기

4. 인수로서의 구조체

모든 구조체는 함수에 인수로서 넘길수 있다. 두가지 실례를 고찰하자. 하나는 Distance구조체의 실례이고 다른 하나는 도형을 표시하는 구조체의 경우이다.

- Distance구조체넘기기

실례 5-5는 Distance형의 인수를 사용하는 함수를 보여준다.

(실례 5-5) 인수로서 구조체넘기기

```
#include <iostream>
using namespace std;
struct Distance
{
  int meters;
  float centies;
};
void Display(Distance);
int main()
{
  Distance d1, d2;
  cout << "메터를 입력하시오:";
  cin >> d1.meters;
  cout << "센치메터를 입력하시오:";
  cin >> d1.centies;
  cout << "\n메터를 입력하시오:";
  cin >> d2.meters;
  cout << "센치메터를 입력하시오:";
  cin >> d2.centies;
  cout << "\nd1=";
  Display(d1);
  cout << "\nd2=";
  Display(d2);
  cout << endl;
  return 0;
}
void Display(Distance dd)
{
  cout << dd.meters << "m " << dd.centies << "㎝";
}
```

이 실례에서 main()부분은 사용자로부터 메터-센치메터형식으로 두개의 거리를 받아들여 두개의 구조체 d1과 d2에 보관한다. 그다음 Distance구조체변수를 인수로 하여 함수 Display()를 호출한다.

함수의 목적은 넘어온 두개의 거리를 표준형식으로 출력하는것이다. 출력은 다음과 같다.

```
메터를 입력하시오: 6
센치메터를 입력하시오: 4
메터를 입력하시오: 5
센치메터를 입력하시오: 4.25
nd1= 6m 4㎝
nd2= 5m 4.25㎝
```

main()안의 함수선언과 함수호출, 함수본체와 선언자에서는 char나 int와 같은 기본형의 변수를 사용할 때와 같이 Distance형의 구조체변수를 인수로서 취급한다.

main()에는 함수 Display()에 대한 호출이 두번 있다. 우선 구조체 d1을 넘기고 다

음에 d2을 넘긴다. 함수 Display()는 Distance형의 구조체형파라메터 dd를 사용한다. 이 구조체변수는 단순변수처럼 main()으로부터 넘어온 구조체값으로 자동적으로 초기화된다. 그다음 Display()안의 명령문들은 dd.meters와 dd.centies식을 사용하여 보통과 같은 방법으로 dd의 성원들을 호출할수 있다. 그림 5-4는 함수에 구조체를 인수로서 넘기는 방법을 보여준다.

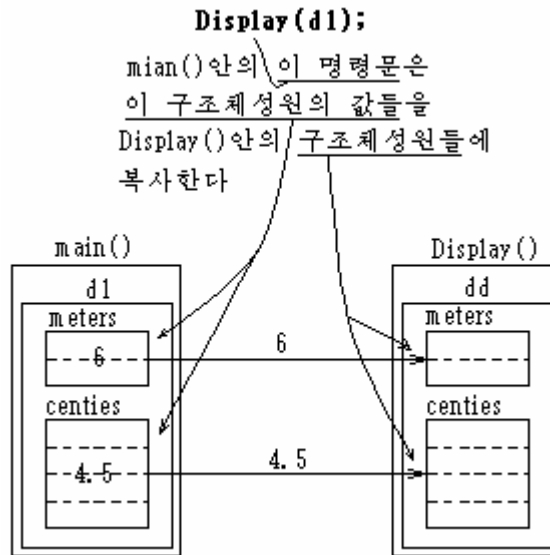



그림 5-4. 인수로서 넘긴 구조체

단순변수와 마찬가지로 Display()의 구조체파라메터 dd는 함수에 넘어온 인수 d1, d2과 같지 않다. 따라서 Display()는 d1와 d2에 영향을 주지 않고 dd를 수정할수 있다. 즉 Display()가 명령문

```
dd.meters = 2;
dd.centies = 3.25;
```

을 포함한다면 이것은 main()의 d1 혹은 d2에 아무런 영향도 주지 않는다.

- Rectangle구조체넘기기

함수에 인수를 넘기는 다른 실례로서 실례 5-6을 고찰하자.

이 실례에서는 Rectangle이라고 부르는 구조체를 선언한다. 직4각형은 폭과 높이를 요소로 가진다.

(실례 5-6) 인수로서 구조체넘기기

```
#include <iostream>
using namespace std;
struct Rectangle
{
  int width;
  int height;
};
void RectArea(Rectangle r)
{
```

```
    int area = r.height * r.width;
    cout << "넓이=" << area << endl;
  }
  int main()
  {
    Rectangle r1 = { 15, 7 };
    Rectangle r2 = { 41, 12 };
    Rectangle r3 = { 65, 18 };
    RectArea(r1);
    RectArea(r2);
    RectArea(r3);
    return 0;
  }
```

Rectangle형의 변수 r1, r2, r3은 각이한 모임값들로 초기화된다. 실례로 r1은

```
Rectangle r1 = { 15, 7 };
```

직4각형들을 창조하고 초기화한 다음에는 RectArea()함수를 세번 호출하여 면적을 계산한다. 프로그람의 실행결과는 다음과 같다.

```
넓이= 105
넓이= 492
넓이= 1170
```

5. 선언안의 이름

선언에 자료형과 함께 의미있는 이름을 삽입하여 함수선언을 명백히 하는 방법이 있다. 실례로 화면우의 점을 표시하는 함수를 사용한다고 하자. 이때 자료형만 가지는 선언을 쓸수 있다. 즉

```
void DisplayPoint(int, int);// 선언
```

또한 다른 방법은

```
void DisplayPoint(int horz, int vert);    // 선언
```

이다.

이 두가지 선언은 번역프로그람에 꼭같은 의미를 준다. 그러나 첫째 방법 즉 (int, int)는 인수들이 수평자리표와 수직자리표용이라는것을 암시할수 없다. 둘째 방법의 우점은 프로그람작성자에게 명백하다. 즉 이러한 선언은 누구든지 함수를 호출할 때 인수를 정확히 사용할수 있게 한다.

선언안의 이름은 함수를 호출할 때 사용하는 이름에 어떤 영향도 주지 않는다. 어떤 인수이름을 사용하는가 하는것은 사용자의 자유이다. 즉 다음과 같이 할수 있다.

```
DisplayPoint(x, y);
```

프로그람을 명백하게 하려고 할 때 이름 + 자료형의 수법을 사용한다.

제 3 절. 함수로부터 값의 돌려주기

함수는 실행을 끝낼 때 호출측프로그람에 하나의 값을 돌려줄수 있다. 보통 돌림값은 함수가 해결한 문제에 대한 답으로 된다. 실례 5-7은 mile값이 주어졌을 때 그것을 km값으로 변환하여 돌려주는 함수를 보여준다.

(실례 5-7) 함수로부터 값을 돌려주기

```
#include <iostream>
using namespace std;
float MilesToKms(float);
int main()
{
   float miles, kms;
   cout << "마일수를 입력하시오:";
   cin >> miles;
   kms = MilesToKms(miles);
   cout << "거리는 " << kms << "km\n";
   return 0;
}
float MilesToKms(float miles)
{
   float kilometers = 1.892 * miles;
   return kilometers;
}
```

프로그람의 실행결과는 다음과 같다.

```
마일수를 입력하시오: 100
거리는 189.2km
```

함수가 값을 돌려줄 때에는 값의 자료형을 지적하여야 한다. 함수선언은 선언과 정의의 함수이름앞에 자료형(실례로 float)을 배치함으로써 돌림값의 형을 지정한다. 앞의 실례들에서는 함수가 돌림값을 돌려주지 않으므로 돌림값형은 void이다.

우의 실례에서 MilesToKms()함수는 float를 돌려주므로 다음과 같이 선언한다.

```
float MilesToKms(float);
```

여기서 처음에 있는 float는 돌림값형을 지정한다. 괄호안의 float는 MilesToKms()에로 넘어오는 인수도 역시 float형이라는것을 지정한다.

함수가 값을 돌려줄 때 함수에로의 호출

```
MilesToKms(miles);
```

은 함수가 돌려주는 값을 가지는 식으로 고찰된다. 이 식을 임의의 다른 변수처럼 취급할수 있다. 즉 이 식을 대입명령문에서 사용할수 있다.

```
kms = MilesToKms(miles);
```

이것은 변수 kms에 MilesToKms()가 돌려주는 값을 대입하게 한다.

1. return명령문

함수 MilesToKms()는 mile로 표시된 인수를 넘겨받아서 그것을 파라메터 miles에 보관한다. 그리고 miles값에 상수를 곱하여 km로 된 거리를 계산하여 결과를 변수 kilometers에 보관한다. 그다음 이 변수값은 return명령문에 의하여 호출측프로그람에 돌아온다.

```
return kilometers;
```

main()과 MilesToKms()는 둘다 km변수 즉 main()에는 kms, MilesToKms()에서는 kilometers를 가지고있다. 함수로부터 돌아올 때 km값은 kms에 복사된다. 호출측 프로그람은 함수안에 있는 km변수를 호출하는것이 아니라 오직 값만 받아들인다. 이 처리를 그림 5-5에 주었다.

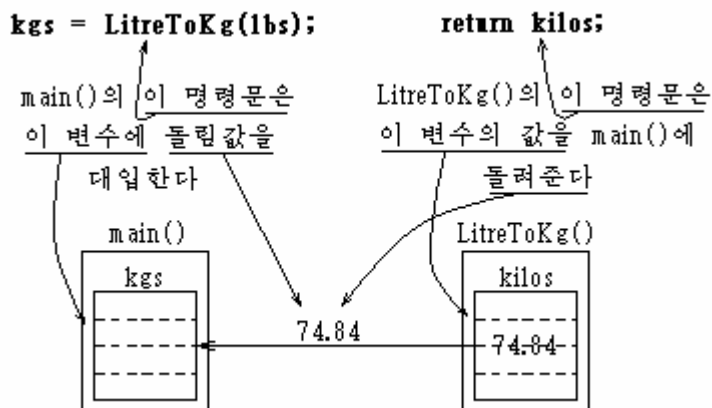



그림 5-5. 값의 돌려주기

여러개의 인수를 함수에 넘길수 있지만 함수로부터는 오직 하나의 인수만 받아들일수 있다. 이것은 함수로부터 여러개의 정보를 돌려주어야 할 때 제한으로 된다. 함수로부터 여러개의 값을 돌려주는 방법이 있다. 그것은 참고에 의하여 인수를 넘기는 방법이다.

함수선언에는 항상 함수의 돌림값형을 포함하여야 한다. 함수가 아무것도 돌려주지 않을 때 예약어 void를 사용한다. 선언에서 돌림값형을 쓰지 않으면 번역프로그람은 함수가 int값을 돌려주는것으로 가정한다. 실례로 선언

```
SomeFunc(); // 선언 … 돌림값형을 int로 가정한다
```

은 번역프로그람에 SomeFunc()가 int형의 돌림값을 가진다고 알린다.

이것은 초기에 C언어에서 규정된것이다. 그러나 기정형을 사용하지 않고 돌림값을 명백히 지정하는것이 리해하기 쉽다.

- 불필요한 변수의 삭제

실례 5-7에서는 입력명령문에서 실제로 필요없는 변수를 여러개 사용한다. 실례 5-7을 변경한 실례 5-8의 프로그람은 함수를 포함하는 식들을 변수의 위치에서 사용하는 방법을 보여준다.

(실례 5-8) 필요없는 변수의 삭제

```
#include <iostream>
using namespace std;
float MilesToKms(float);
int main()
{
  float miles;
  cout << "마일수를 입력하시오:";
  cin >> miles;
  cout << "거리는 " << MilesToKms(miles) << "km\n";
  return 0;
}
float MilesToKms(float miles)
{
  return 1.892 * miles;
}
```

이 실례의 main()에서는 실례 5-7로부터 변수 kms를 삭제하고 그대신에 함수 MilesToKms(miles)를 직접 cout명령문에 삽입한다. 즉

```
cout << "거리는 " << MilesToKms(miles) << "km\n";
```

또한 MilesToKms()함수에서는 변수 kilometers를 사용하지 않고 식 1.892 * miles를 직접 return명령문에 삽입한다.

```
return 1.892 * miles;
```

계산이 진행되면 결과값이 호출측프로그람에로 마치도 변수값처럼 돌아온다.

프로그람작성자들은 흔히 return명령문안의 식을 괄호안에 넣군 한다. 즉

```
return (1.892 * miles);
```

번역프로그람이 요구하지 않아도 식안에 여분의 괄호를 넣으면 프로그람을 읽기 쉽다.

경험있는 C++프로그람작성자에게는 실례 5-7보다 실례 5-8의 형식이 더 나을것이다. 그러나 실례 5-8은 비전문가인 경우에 리해하기 힘들다.

2. 구조체변수의 돌려주기

구조체는 함수의 인수 또는 돌림값으로 사용할수도 있다.

실례 5-9는 Distance형변수를 추가하고 그 형의 값을 돌려준다.

(실례 5-9) 구조체 돌려주기

```
#include <iostream>
using namespace std;
struct Distance
{
  int meters;
  float centies;
};
Distance AddDist(Distance, Distance);
```

```
void Display(Distance);
int main()
{
  Distance d1, d2, d3;
  cout << "메터를 입력하시오:"; cin >> d1.meters;
  cout << "센치메터를 입력하시오:"; cin >> d1.centies;
  cout << "\n메터를 입력하시오:"; cin >> d2.meters;
  cout << "센치메터를 입력하시오:"; cin >> d2.centies;
  d3 = AddDist(d1, d2);
  cout << endl; Display(d1);
  cout << " + "; Display(d2); cout << " = "; Display(d3);
  return 0;
}
Distance AddDist(Distance dd1, Distance dd2)
{
  Distance dd3;
  dd3.centies = dd1.centies + dd2.centies;
  dd3.meters = 0;
  if(dd3.centies >= 100.0)
  {
     dd3.centies -= 100.0;
     dd3.meters++;
  }
  dd3.meters += dd1.meters + dd2.meters;
  return dd3;
}
void Display(Distance dd)
{
  cout << dd.meters << "m " << dd.centies << "㎝";
}
```

프로그람은 사용자로부터 메터-센치메터형의 길이를 두개 읽어들이고 그것들을 함수 AddDist()에 의하여 더하고 실례 5-5에서 소개한 Display()함수에 의하여 결과를 표시한다. 프로그람의 출력은 다음과 같다.

```
메터를 입력하시오: 4
센치메터를 입력하시오: 5.5
메터를 입력하시오: 5
센치메터를 입력하시오: 6.5
4m 5.5㎝ + 5m 6.5㎝ = 9m 12.0㎝
```

프로그람의 main()에서는 함수 AddDist()를 호출하여 각각 Distance형구조체로 표시된 두개의 길이를 더한다.

```
d3 = AddDist(d1, d2);
```

이 함수는 d1과 d2의 값을 더하여 돌려주며 그 결과는 main()의 d3에 대입된다.

AddDist()는 계산결과를 보관하기 위하여 Distance형의 새 변수를 창조하여야 한다. 그것은 간단히

```
return dd1 + dd2;
```

와 같은 식을 돌려줄수 없기때문이다. 두개 구조체의 더하기는 여러 단계에 걸쳐 수행된다. dd3의 개별적인 성원들의 값을 계산하고 그다음 명령문

```
return dd3;
```

에 의하여 호출측프로그람에 돌아온다. 그 결과는 호출측프로그람의 d3에 대입된다.

이 프로그람은 구조체를 돌림값으로 사용하는 방법을 보여주는것과 함께 같은 프로그람에서 사용하는 두 함수를 보여준다. 함수들은 임의의 순서로 배렬할수 있다. 유일한 규칙은 함수에 대한 호출이 있기전에 함수선언이 있어야 한다는것이다.

제 4 절. 참고인수

참고(reference)는 변수에 별명(alias)을 제공한다. 참고의 가장 중요한 사용의 하나는 함수에 인수를 넘기는 경우이다.

이미 값에 의하여 넘기는 함수인수의 실례들을 보았다. 값에 의하여 인수를 넘길 때 호출되는 함수는 같은 형의 새 변수를 창조하고 인수값을 거기에 복사한다. 이 함수는 호출측프로그람안에 있는 원시변수를 호출할수 없고 다만 원시변수의 사본을 창조한다. 값에 의한 인수넘기기는 함수가 호출측프로그람안에 있는 원시변수를 수정할 필요가 없을 때 쓸모있다. 사실상 이것은 함수가 원시변수를 변경하지 않는다는것을 담보한다.

참고에 의한 인수넘기기는 다른 기구를 사용한다. 즉 함수에로 값을 넘길 대신에 호출측프로그람안에 있는 원시변수에로의 참고를 넘긴다. 참고는 실제상 넘기는 변수의 기억주소이다.

참고에 의한 넘기기의 중요한 우점은 함수가 호출측프로그람에 있는 실제 변수를 호출할수 있다는것이다. 이것은 함수로부터 호출측프로그람에로 한개이상의 값을 넘기기 위한 기구를 제공해준다.

1. 참고에 의한 기본자료형 넘기기

다음의 실례 5-10은 참고에 의해 넘기는 단순변수를 보여준다.

(실례 5-10) 참고에 의한 넘기기

```
#include <iostream>
using namespace std;
void IntFrac(float, float&, float&);
int main()
{
   float number, intPart, fracPart;
   do
   {
      cout << "\n실수를 입력하시오: ";
      cin >> number;
```

```
        IntFrac(number, intPart, fracPart);
        cout << "옹근수부 = " << intPart
            << " 소수부 = " << fracPart << endl;
    } while(number != 0.0);
    return 0;
}
void IntFrac(float n, float& intp, float& fracp)
{
    long temp = static_cast<long>(n);
    intp = static_cast<float>(temp);
    fracp = n - intp;
}
```

이 실례에서 main()은 사용자로부터 float형의 수값을 받아들여 옹근수와 소수부로 가른다. 즉 사용자가 입력한 수가 12.456이면 옹근수부가 12.0, 소수부가 0.456이다. 이 두개의 수값을 얻기 위하여 main()은 함수 IntFrac()를 호출한다. 아래에 프로그람의 출력이 있다.

```
실수를 입력하시오: 99.44
옹근수부 = 99 소수부 = 0.44
```

일부 번역프로그람들은 0.440002와 같이 소수부에서 근사한 수값을 발생시킬수 있다. 그것은 번역프로그람의 변환루틴에서의 오유로서 무시할수 있다.

IntFrac()함수는 강제형변환에 의하여 파라메터 n에 넘긴 수를 long형변수로 변환하여 옹근수를 얻는다. 즉 다음의 식을 사용한다.

```
long temp = static_cast<long>(n);
```

이것은 n의 소수부를 잘라버린다. 그러므로 옹근수형은 옹근수부에만 보관되고 그 결과는 다른 강제형변환

```
intp = static_cast<float>(temp);
```

에 의해 float형으로 다시 변환된다.

소수부는 단순히 옹근수부보다 작은 원시수이다.(서고함수 fmod()를 사용할수 있다.)

그러면 IntFrac()함수가 옹근수부와 소수부를 얻어서 main()에 어떻게 넘겨주는가?

한개의 값이라면 return명령문을 사용하면 되는데 값이 두개이다. 이 문제를 참고인수에 의하여 해결한다. 여기에 함수선언자가 있다.

```
void IntFrac(float n, float& intp, float& fracp)
```

참고인수는 자료형뒤에 앰퍼센트기호 &를 붙여 지정한다.

```
float& intp
```

&는 intp가 어떤 변수용의 인수로서 넘기는 별명이라는것을 의미한다. 다시 말하여 IntFrac()함수에서는 intp라는 이름을 사용하여 main()에 있는 intPart를 참고한다.

&는 참고를 의미하므로

```
float& intp
```

는 intp가 float변수에로의 참고라는것을 의미한다. 마찬가지로 fracp는 fracPart의 별

명 즉 fracPart에로의 참고이다.

함수선언은 정의안에서 &의 사용을 반영한다.

```
void IntFrac(float, float&, float&);
```

함수선언에서도 정의에서처럼 참고에 의해 넘어오는 인수앞에 &를 놓는다.

함수호출에서는 &를 쓰지 않는다. 즉

```
IntFrac(number, intPart, fracPart);
```

함수호출에서는 인수를 참고에 의해 넘기는지 값에 의해 넘기는지 모른다.

intPart와 fracPart는 참고에 의해 넘겨지고 변수 number는 값에 의해 넘겨진다. intp와 intPart, fracp와 fracPart는 같은 기억장소들을 가지지만 다른 이름을 가진다. 다른 항목 number는 IntFrac()함수가 수정할 필요가 없으므로 값에 의하여 넘긴다. 그림 5-6은 참고인수의 동작을 보여준다.(주소연산자 &와 참고연산자 &는 같지 않다.)

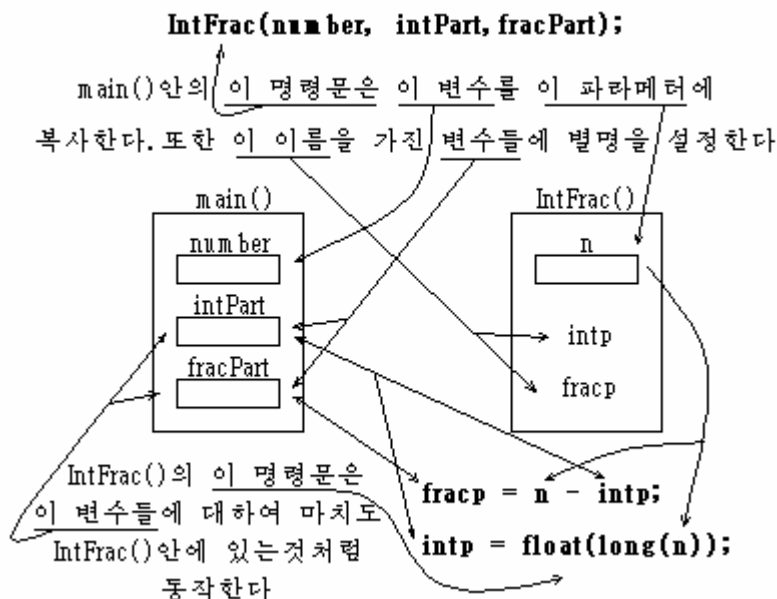



그림 5-6. 참고에 의한 넘기기

- 참고에 의한 복잡한 넘기기

참고에 의해 한개 인수를 넘기는 복잡한 실례가 있다. 실례 5-11에서는 여러쌍의 수들에 대하여 작은 수가 큰수보다 앞에 놓이는가를 조사한다. 이를 위하여 함수 Order()를 호출한다. Order()는 참고에 의해 넘어온 두 수를 검사하여 첫째 수가 둘째 수보다 크면 이 수들을 서로 교체한다.

(실례 5-11) 참고에 의하여 넘긴 두개 인수의 정렬

```
#include <iostream>
using namespace std;
void Order(int&, int&);
int main()
{
```

```
    int n1 = 99, n2 = 11;
    int n3 = 22, n4 = 88;
    Order(n1, n2);
    Order(n3, n4);
    cout << "n1=" << n1 << endl;
    cout << "n2=" << n2 << endl;
    cout << "n3=" << n3 << endl;
    cout << "n4=" << n4 << endl;
    return 0;
}
void Order(int& numb1, int& numb2)
{
    if(numb1 > numb2)
    {
        int temp = numb1;
        numb1 = numb2;
        numb2 = temp;
    }
}
```

main()에는 두 쌍의 수들이 있는데 첫째 쌍은 순서화되여있지 않고 둘째 쌍은 순서화되여있다. 매개 쌍에 대하여 main()함수를 한번씩 호출하고 그 수들을 모두 출력한다. 출력은 첫 쌍은 교체되고 둘째 쌍은 교체되지 않는다는것을 보여준다. 프로그람의 출력은 다음과 같다.

```
n1=11
n2=99
n3=22
n4=88
```

Order()함수에서는 첫째 변수를 numb1, 둘째 변수를 numb2로 한다. numb1이 numb2보다 크면 함수는 numb1을 temp에, numb2를 numb1에, temp를 numb2에 보관한다. numb1과 numb2는 인수를 넘기는데 쓰이는 서로 다른 이름이다. 이 경우에 처음의 함수호출에서는 n1와 n2, 둘째 호출에서는 n3과 n4이다. 함수는 호출측프로그람의 원시인수들의 순서를 검사하고 필요하다면 그것들을 교체한다.

이러한 방법으로 참고인수를 사용하는것은 원격조종조작의 한 부류이다. 호출측프로그람은 함수가 호출측프로그람의 변수들에 대하여 조작하게 하고 함수는 이 변수들의 실제이름은 모르지만 값들을 변경한다.

2. 참고에 의한 구조체넘기기

참고에 의하여 구조체를 기본자료형처럼 넘길수 있다. 실례 5-12는 Distance형값들에 대한 비례변환을 처리한다. 비례변환이란 한묶음의 거리들에 어떤 결수를 곱하는것을 말한다. 거리가 6m 8cm이고 비례결수가 0.5이면 새 거리는 3m 4cm이다. 그러한 변환은 일정한 비률로 건물을 신축하는 경우에 건물의 모든 치수들에 적용할수 있다.

(실례 5-12) 참고에 의한 구조체넘기기

```
#include <iostream>
using namespace std;
struct Distance
{
  int meters;
  float centies;
};
void Scale(Distance&, float);
void Display(Distance);
int main()
{
  Distance d1 = { 12, 6.5 };
  Distance d2 = { 10, 5.5 };
  cout << "d1="; Display(d1);
  cout << "\nd2="; Display(d2);
  cout << endl;
  Scale(d1, 0.5);
  Scale(d2, 0.25);
  cout << "d1="; Display(d1);
  cout << "\nd2="; Display(d2);
  cout << endl;
  return 0;
}
void Scale(Distance& dd, float factor)
{
  float centies = (dd.meters * 100 + dd.centies) * factor;
  dd.meters = static_cast<int>(centies / 100);
  dd.centies = centies - dd.meters * 100;
}
void Display(Distance dd)
{
  cout << dd.meters << "m " << dd.centies << "cm";
}
```

실례에서는 두개의 거리변수 d1과 d2을 어떤 값으로 초기화하고 표시한다. 그다음 Scale()함수를 호출하여 d1에 0.5, d2에 0.25를 곱한다. 끝으로 그 거리값들을 표시한다. 프로그람의 출력은 다음과 같다.

```
d1=12m 6.5cm
d2=10m 5.5cm
d1=6m 3.25cm
d2=2m 51.375cm
```

여기에 Scale()에로의 함수호출이 두개 있다.

```
Scale(d1, 0.5);
Scale(d2, 0.25);
```

처음의 호출은 d1에 0.5를 곱하고 둘째 호출은 d2에 0.25를 곱한다. 이러한 변경은 d1과 d2에 대하여 직접 진행된다. 함수는 아무것도 돌려주지 않고 Distance인수에

직접 조작하며 인수는 Scale()에로의 참고에 의해 함수에 넘어온다.

호출측프로그람에서 유일한 한개값이 변경되므로 값에 의해 인수를 넘기고 값을 돌려주도록 함수를 다시 쓸수 있다. 이와 같은 함수호출은 다음과 같다.

```
d1 = scale(d1, 0.5);
```

그러나 이것은 필요없다.

3. 참고에 의한 넘기기에서 주의사항

참고에 의한 인수넘기기는 Pascal에서도 가능하고 새로운 판의 BASIC에서도 가능하다. C에는 참고가 없으므로 지적자를 류사한 목적에 사용한다. 참고인수는 단순변수는 물론 객체의 호출을 포함하고있는 모든 경우에 융통성을 제공하기 위하여 C++에 도입되였다.

값에 의한 넘기기와 참고에 의한 넘기기외에 함수에 인수를 넘기는 세번째 방법은 지적자를 사용하는것이다.

제 5 절. 재정의된 함수

재정의된 함수(overloaded functoin)는 함수에 보내는 자료의 종류에 따라서 각이한 동작을 처리한다. 재정의된 함수는 어떤 종류의 자료에 대해서는 어떤 조작을 수행하지만 다른 종류의 자료에 대해서는 다른 조작을 수행한다.

1. 인수의 개수

실례 5-1의 StarLine함수와 실례 5-3의 RepChar()함수를 상기해보자. StarLine()함수는 한행에 별표를 45개 출력하고 RepChar()는 함수호출에서 주어지는 문자와 행길이를 사용한다. 또한 세번째 함수로서 CharLine()함수는 항상 45개 문자를 출력하고 호출측프로그람이 출력할 문자를 지정한다.

세개의 함수 StarLine(), RepChar(), CharLine()은 서로 류사한 동작을 하지만 서로 다른 이름을 가지고있다. 작성자들이 이러한 함수를 사용하려면 세개의 함수가 있어야 하며 프로그람에서 함수를 참고하려면 자모순으로 입력되여있는 경우 세 곳에서 찾아야 한다.

함수들이 비록 다른 인수를 가지고있지만 함수들의 이름은 모두 같게 하는것이 관례이다.

실례 5-13에서 이것을 실현한다.

(실례 5-13) 함수재정의

```
#include <iostream>
using namespace std;
void RepChar();
```

```
void RepChar(char);
void RepChar(char, int);
int main()
{
   RepChar();
   RepChar('=');
   RepChar('+', 30);
   return 0;
}
void RepChar()
{
   for(int j=0; j<45; j++)
      cout << '*';
   cout << endl;
}
void RepChar(char ch)
{
   for(int j=0; j<45; j++)
      cout << ch;
   cout << endl;
}
void RepChar(char ch, int n)
{
   for(int j=0; j<n; j++)
      cout << ch;
   cout << endl;
}
```

이 프로그람은 세개 행의 문자렬을 출력한다.

```
*********************************************
=============================================
++++++++++++++++++++++++++++++
```

처음 두 행은 45문자이고 세번째 행은 30문자이다. 프로그람에는 같은 이름을 가지는 함수가 세개 들어있다. 즉 세개의 선언, 세개의 함수호출, 세개의 함수정의가 있다. 번역프로그람은 함수를 판별할 때 인수의 개수와 그 자료형을 사용한다. 다시말하여 선언

```
void RepChar();
```

에는 인수가 없고 char를 하나의 인수로 가지는 선언

```
void RepChar(char);
```

과 전혀 다른 함수를 서술한다. 또한 char형과 int형의 인수를 가지는 선언

```
void RepChar(char, int);
```

과도 완전히 다른 함수로 서술된다.

번역프로그람은 이름이 같지만 인수개수가 서로 다른 여러개의 함수를 발견하면 작성자가 오유를 범했는가를 조사한다. 그리고 개별적인 함수를 그 함수계렬의 정의로 설정한다. 그중 어느 함수가 호출되는가는 호출할 때 제공되는 인수의 개수에 의존한다. 그림 5-7은 그 과정을 보여준다.

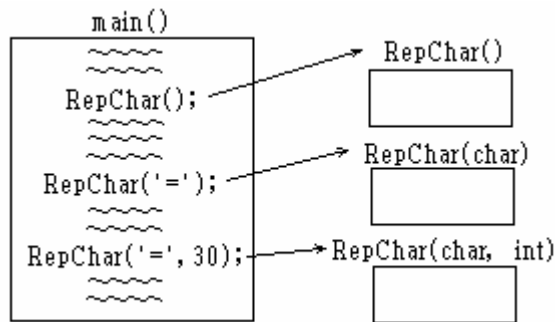



그림 5-7. 재정의된 함수들

2. 인수의 형

실례 5-13에서는 이름은 같지만 각이한 개수의 인수를 가지는 여러개의 함수들을 창조한다. 또한 번역프로그람은 인수개수와 그 형의 차이를 보고 재정의된 함수들을 구별할수 있다. 실례 5-14에서는 메터와 센치메터형식으로 거리를 표시하기 위해 재정의된 함수들을 사용한다. 함수에 대한 유일한 인수는 Distance형의 구조체이거나 float형의 단순변수이다. 인수형에 따라 서로 다른 함수들이 사용된다.

(실례 5-14) 함수의 재정의

```
#include <iostream>
using namespace std;
struct Distance
{
   int meters;
   float centies;
};
void Display(Distance);
void Display(float);
int main()
{
   Distance d1;
   float d2;
   cout << "\n메터를 입력하시오: "; cin >> d1.meters;
   cout << "센치메터를 입력하시오: "; cin >> d1.centies;
   cout << "센치로 환산한 거리를 입력하시오: "; cin >> d2;
   cout << "\nd1="; Display(d1);
   cout << "\nd2="; Display(d2);
   cout << endl;
   return 0;
}
void Display(Distance dd)
{
   cout << dd.meters << "m " << dd.centies << "cm";
}
```

```
void Display(float dd)
{
  int meters = static_cast<int>(dd / 100);
  float centies = dd - meters * 100;
  cout << meters << "m " << centies << "㎝";
}
```

사용자는 두개의 거리를 입력한다. 처음에는 메터와 센치메터를 따로따로 입력하고 다음에는 하나의 센치메터를 입력한다. 이 프로그람은 재정의된 함수 Display()를 호출하여 첫째 거리용으로 Distance형의 값, 둘째 거리용으로 float형의 값을 표시한다. 여기에 프로그람과의 대화가 있다.

```
메터를 입력하시오: 5
센치메터를 입력하시오: 10.5
센치로 환산한 거리를 입력하시오: 176.5
d1=5m 10.5cm
d2=1m 76.5cm
```

Display()의 서로 다른 판이 서로 류사한 일을 하지만 코드는 완전히 다르다. 메터와 센치메터를 받아들이는 판에서는 결과를 표시하기 전에 메터나 센치메터로 변환되여야 한다.

재정의된 함수들은 많은 함수이름의 기억으로부터 해방해줌으로써 작성자의 작업을 단순화한다. 재정의를 사용하지 않을 때 제기되는 복잡성에 대하여 수의 절대값을 구하는 C++의 서고함수들을 고찰하자. 서고함수들은 C++는 물론 C(재정의를 허용하지 않는다.)와 작업해야 하므로 매개 자료형에 대하여 절대값함수를 따로따로 제공하고있다. 즉 int형에 abs(), 복소수형에 cabs(), double형에 fabs(), long형에 labs()을 제공한다. C++에서는 하나의 이름 abs()가 이 자료형 모두에 대하여 동작한다.

제 6 절. inline함수와 기정인수

1. inline함수

함수에로의 모든 호출이 같은 코드를 실행하게 하므로 함수는 기억공간을 절약한다고 말할수 있다. 즉 함수본체를 기억기에 반복보관할 필요가 없다. 번역프로그람은 함수호출과 만나면 보통 그 함수에로의 이행을 생성한다. 그리고 함수의 끝에서 함수호출뒤에 오는 명령문에로 되돌아온다. (그림 5-1 참고)

함수호출은 기억공간을 절약하는 한편 보충적인 시간을 소비한다. 함수에로의 이행명령(기호언어명령 CALL), 등록기보관명령, 인수가 있는 경우 호출측프로그람안의 탄창에 인수들의 밀어넣기와 탄창으로부터 함수에로 인수들을 꺼내는 명령, 등록기회복명령, 호출측프로그람에로의 귀환명령이 있어야 한다. 또한 돌림값도 취급해야 한다. 이 모든 명령들은 프로그람의 속도를 떨군다.

간단한 함수호출에서는 실행시간을 줄이기 위하여 호출측프로그람의 코드뒤에 곧 함수본체의 코드를 삽입할수 있다. 즉 원천파일에서 함수를 호출할 때마다 함수에로의 이행대신에 함수의 실례코드가 실행된다. 함수와 inline코드사이의 차이를 그림 5-8에 주었다.

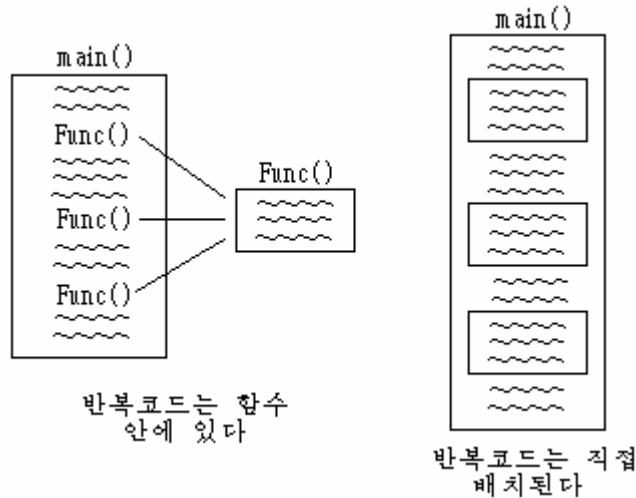



그림 5-8. 함수와 inline코드

그러나 긴 반복코드는 일반함수로 하는것이 더 좋다. 기억공간의 예약은 실행속도에서 비교적 적은 손실을 가져온다. 그러나 짧은 코드를 일반함수로 하면 기억공간이 적게 예약되고 큰 함수에서처럼 보충적인 시간을 소비하게 한다.

사실상 함수가 아주 짧으면 그 호출에 필요한 명령들은 함수본체안에 있는 명령들처럼 많은 공간을 차지하므로 시간뿐아니라 공간도 랑비한다.

이러한 경우에 필요할 때마다 한 묶음의 명령들을 삽입하여 프로그람안에서 단순히 코드를 반복할수 있다. 같은 코드를 반복삽입하는데서 난관은 프로그람의 조직화와 함수사용으로부터 명백성을 완전히 잃는것이다. 프로그람의 속도는 더 빨라지고 적은 공간을 소비하지만 프로그람은 더 길고 복잡해진다.

바로 이에 대한 해결방도가 inline함수이다. inline함수는 원천파일에 표준함수처럼 쓰이지만 함수객체는 inline코드로 번역된다. 원천파일은 잘 조직화되고 읽기 쉽게 그대로 유지되지만 함수는 여전히 개별적인 실체이다. 그러나 프로그람을 번역할 때 함수본체는 실제로 함수호출이 발생하는 프로그람안에로 삽입된다.

명령문이 한개인 아주 짧은 함수는 inline으로 할수 있는 후보이다. 여기에 실례 5-8을 변경한 실례 5-15가 있다. 여기서는 MilesToKm()함수를 inline으로 한다.

(실례 5-15) inline함수

```
#include <iostream>
using namespace std;
inline float MilesToKm(float miles)
{
```

```
    return miles * 1.852;
}
int main()
{
    float mi;
    cout << "\n마일로 된 거리를 입력하시오: ";
    cin >> mi;
    cout << "km로 환산한 거리는 " << MilesToKm(mi) << endl;
    return 0;
}
```

함수를 간단히 inline으로 할수 있다. 즉 함수정의앞에 예약어 inline을 놓는다.

번역프로그람은 함수가 inline으로 하기에는 너무 길다고 결정하면 그 함수를 일반함수로서 번역한다.

C++프로그람작성자는 C의 #define마크로대신에 inline함수를 널리 사용한다. 이것들은 같은 목적으로 사용하지만 inline함수가 #define마크로보다 형검사를 더 잘 제공하고 괄호로 인한 주의를 요구하지 않는다.

2. 기정인수

함수는 인수를 모두 지정하지 않고도 호출할수 있다. 이것은 아무 함수에 대해서나 가능한것은 아니고 함수선언에서 지정하지 않는 인수의 기정값을 제공해야 한다.

실례 5-16은 실례 5-13을 변경한것으로서 기정인수(default argument)를 보여준다. 실례 5-13에서는 같은 이름을 가진 세개의 함수를 각이한 개수의 인수들을 조종하는데 사용한다. 다음 실례 5-16은 다른 방법으로 같은 효과를 얻는다.

(실례 5-16) 기정인수

```
#include <iostream>
using namespace std;
void RepChar(char = 'x', int=45);
int main()
{
    RepChar();
    RepChar('=');
    RepChar('+', 30);
    return 0;
}
void RepChar(char ch, int n)
{
    for(int j=0; j<n; j++)
        cout << ch;
    cout << endl;
}
```

이 프로그람에서 함수 RepChar()는 두개의 인수를 가진다. RepChar()함수는 main()으로부터 세번 호출된다. 처음에는 인수없이, 두번째는 한개인수, 세번째는 두개 인수로서 호출된다.

그러면 처음 두개의 호출이 어떻게 동작하는가?

호출된 함수는 기정인수를 제공하므로 호출하는 프로그람이 그 인수를 제공하지 않으면 기정인수가 쓰인다. 기정인수는 RepChar()의 선언에서 주어진다.

```
void RepChar(char = 'x', int=45);
```

기정인수는 형이름뒤에 같기기호와 함께 놓는다. 또한 변수이름을 사용할수 있다. 즉

```
void RepChar(char ch = 'x', int n =45);
```

함수호출에서 한개 인수를 생략하면 그것을 마지막인수로 가정한다. RepChar()함수는 ch에 유일인수값을 대입하고 n으로서 기정값 45를 사용한다.

둘째 인수를 생략하면 함수는 ch에 기정값 'x', n에 기정값 45를 대입한다. 이리하여 함수에 대한 세번의 호출은 매번 인수개수가 다르지만 모두 동작한다.

생략하는 인수는 인수목록의 끝에 있는 꼬리인수이다. 인수목록의 중간에 있는 임의의 인수들을 생략할수 없다. 이것은 타당하다.

번역프로그람이 중간에서 생략한 인수들의 의미를 어떻게 알겠는가?

만일 기정값을 제공하지 않는 함수를 인수없이 호출하면 오유를 경고한다.

기정인수는 대체로 늘 같은 값을 가지는 인수들을 사용하는데 효과가 있다. 또한 프로그람을 쓴 다음 프로그람작성자가 다른 인수를 추가하여 함수의 능력을 높이려는 경우에 효과가 있다. 기정인수는 현존인수호출이 이전의 인수들을 계속 보존하면서 새 함수호출을 더 사용하려는 경우에 요구된다.

제 7 절. 변수와 기억등급

이제는 함수를 알고있으므로 변수와 함수들의 호상작용과 관련한 C++의 특성인 기억등급을 리해할 때가 되였다. 변수의 기억등급(class)은 프로그람의 어느 부분에서 변수를 호출할수 있고 변수가 얼마나 오래동안 존재하는가를 결정한다. 세개의 기억등급 즉 자동, 외부, 정적등급을 가진 변수들을 고찰하자.

1. 자동변수

실례들에서 사용한 거의 모든 변수들은 그것을 사용하는 함수안에서 정의되였다. 즉 변수정의는 함수본체안에서 하였다.

```
void SomeFunc()
{
  int someVar;
```

```
    float otherVar;
    // 기타 명령문들
}
```

변수는 main()이나 다른 함수안에서 정의할수 있으며 그 효과는 main()에서와 비슷하다. 함수본체안에서 정의한 변수들은 자동변수이다. 사실상 예약어 auto는 자동변수지정에 쓰인다. 즉 다음과 같이 쓸수 있다.

```
void SomeFunc()
{
    auto int someVar;
    auto float otherVar;
    // 기타 명령문들
}
```

그러나 auto는 기정이므로 auto예약어를 사용할 필요는 거의 없다. 함수안에서 정의된 변수들은 자동변수이다.

여기서 자동변수의 중요한 두가지 특성인 수명과 보임성을 고찰하자.

1) 수명

자동변수는 그것이 정의된 함수가 호출될 때까지 창조되지 않는다. 더 정확히 말하면 임의의 코드블로크안에서 정의한 변수들은 그 블로크를 실행할 때까지 창조되지 않는다. 우에서 정의한 프로그람부분에서 변수 someVar와 otherVar는 SomeFunc()함수가 호출될 때까지 존재하지 않는다. 즉 기억기안에 값들을 보관하는 위치가 없으며 그것들은 정의되지 않는다. 조종이 SomeFunc()에로 넘어올 때 비로소 변수들이 창조되고 기억공간이 설정된다. 후에 SomeFunc()함수로부터 조종이 호출측프로그람에로 돌아올 때 변수들은 파괴되고 값을 잃는다. 자동이라는것은 함수가 호출될 때 변수들이 자동적으로 창조되고 함수로부터 돌아갈 때 자동적으로 해체된다는데로부터 유래되였다. 변수의 창조와 해체사이의 시간주기를 수명(lifetime)이라고 한다. 자동변수의 수명은 그것이 정의되는 함수가 실행될 때 시작된다.

변수의 수명을 제한하는것은 기억공간을 절약하기 위해서이다. 함수를 실행하지 않으면 함수가 사용하는 변수들은 요구되지 않는다. 변수들을 삭제하면 다른 함수들이 사용할수 있게 기억기를 해방한다.

2) 보임성

변수의 보임성(visibility)은 프로그람안에서 변수를 호출할수 있는 위치를 서술한다. 보임성은 프로그람의 한부분에 있는 명령문에서만 변수를 참고하게 하고 다른 부분에서는 오유를 통보하게 한다.

또한 리용범위는 보임성을 서술하는데 쓰인다. 변수의 리용범위(scope)는 변수를 볼수 있는 프로그람의 부분이다.

자동변수는 그것이 정의된 함수안에서만 볼수 있고 호출할수 있다. 프로그람에 두개의 함수가 있다고 하자.

```
void SomeFunc()
{
  int someVar;
  float otherVar;
  someVar = 11;
  otherVar = 11;
  nextVar = 12; // 오유: SomeFunc()에서 볼수 없다.
}
void OtherFunc()
{
  int nextVar;
  somevar = 20; // 오유: OtherFunc()에서 볼수 없다.
  otherVar = 21;
  nextVar = 22;
}
```

변수 nextVar는 함수 SomeFunc()에서 볼수 있고 변수 someVar와 otherVar는 OtherFunc()에서 볼수 있다.

변수의 보임성을 제한하는것은 프로그람을 조직화하고 모듈화하기 위해서이다. 어떤 함수안의 변수들을 다른 함수로부터 볼수 없으므로 다른 함수들에 의한 우연한 변경으로부터 안전하다. 이것은 구조화프로그람작성법의 중요한 부분이다. 또한 보임성의 제한은 객체지향프로그람작성법의 중요한 부분이다.

자동변수인 경우 수명과 보임성은 일치하고 자동변수는 그것이 정의되는 함수를 실행할 때에만 존재하고 그 함수안에서만 볼수 있다. 그러나 다른 일부 기억등급에서 보임성과 수명은 같지 않다.

3) 초기화

자동변수가 창조될 때 번역프로그람은 그것을 초기화하지 않는다. 이때 자동변수는 임의의 값으로 시작되며 그 값은 어떤 값인지 모른다. 자동변수를 초기화하려면 초기값을 명백히 써야 한다. 즉

```
int n = 33;
```

일반적으로 자동변수를 국부변수(local variable)라고도 한다. 그것은 자동변수가 정의된 함수안에서만 국부적으로 볼수 있기때문이다.

2. 외부변수

다음으로 중요한 기억등급은 외부기억(external)등급이다. 자동변수는 함수안에 정의되지만 외부변수는 함수밖에서 정의된다. 외부변수는 프로그람안의 모든 함수들에서 볼수 있다. 보통 외부변수를 모든 함수들에서 볼수 있게 하기 위하여 외부변수선언을 프로그람의 앞부분에 놓는다. 외부변수는 프로그람의 모든 함수들에 알려지므로 대역변수(global variable)라고도 한다.

실례 5-17에서는 세개의 함수들이 모두 외부변수를 호출한다.

(실례 5-17) 외부변수

```
#include <iostream>
using namespace std;
#include <conio.h>
char ch = 'a';
void GetAChar();
void PutAChar();
int main()
{
   while(ch != '\r')
   {
      GetAChar();
      PutAChar();
   }
   cout << endl;
   return 0;
}
void GetAChar()
{
   ch = getch();
}
void PutAChar()
{
   cout << ch;
}
```

이 실례에서 함수 GetAChar()는 서고함수 getch()에 의하여 건반으로부터 문자들을 읽어들인다. getch()는 입력한 문자를 화면에 표시하지 않는것을 제외하면 getche()와 같다. 실례에서 둘째 함수 PutAChar()는 화면에 매개 문자를 표시한다. 일반적으로 사용자가 입력한 문자를 표시한다.

```
I'm typing in this line of text
```

이 실례에서 중요한것은 변수 ch가 함수안에서 정의되지 않고 파일의 선두에 있는 첫함수앞에서 정의되는것이다. 이것이 외부변수이다. 프로그람에서 ch의 정의뒤에 있는 어느 함수에서나 이 변수를 호출할수 있으며 실례에서는 모든 함수 즉 main(), GetAChar(), PutAChar()들에서 호출할수 있다. 그러므로 ch의 보임성은 원천파일전체이다.

1) 외부변수의 역할

외부기억등급은 프로그람의 한개이상의 함수들에서 변수를 호출해야 하는 경우에 사용된다. 수속형프로그람에서 외부변수는 대체로 프로그람에서 가장 중요한 변수로 된다. 그러나 1장에서 서술한것처럼 외부변수는 임의의 함수들을 호출할수 있는데로부터 문제가 생긴다. 이때 함수들에서 외부변수를 잘못 호출할수 있다. 그러나 객체지향프로그람작성에서는 외부변수가 요구되지 않는다.

2) 초기화

외부변수를 초기화할수 있다. 즉

```
int exvar = 179;
```

외부변수의 초기화는 파일이 처음으로 적재될 때 수행된다. 외부변수가 프로그람에 의해 명시적으로 초기화되지 않으면 즉 례를 들어

```
int exvar;
```

라고 정의하면 그것이 창조될 때 자동적으로 0으로 초기화된다. (즉 이것은 변수가 창조될 때 초기화되지 않으므로 의미가 없는 우연값을 가지는 자동변수와 다르다.)

3) 수명과 보임성

외부변수는 프로그람의 수명만큼 존재한다. 즉 프로그람이 시작될 때 기억공간이 외부변수용으로 설정되고 프로그람이 끝날 때까지 계속 존재한다.

이미 설명한것처럼 외부변수는 그것이 정의되는 행으로부터 시작하여 그것이 정의되는 파일안에서 볼수 있다. ch와 main()뒤, GetAChar()앞에서 정의한다면 GetAChar()와 PutAChar()에서만 볼수 있고 main()에서는 볼수 없다.

3. 정적변수

정적기억(static)등급을 고찰하자. 여기서는 정적인 자동변수만 고찰한다. 정적외부변수도 있으나 여러 파일로 된 프로그람에서만 의미가 있다.

정적자동변수는 국부변수의 보임성을 가지므로 그것을 포함하는 함수안에서만 볼수 있다. 정적자동변수의 수명은 그것을 포함하는 함수가 처음으로 호출되기 전에는 존재하지 않는것만 제외하면 외부변수의 수명과 류사하다. 그후 정적변수는 프로그람의 수명기간 존재한다.

정적자동변수는 어떤 함수를 실행하지 않는동안 값을 보관해야 할 때 사용된다. 즉 함수에로의 호출들사이에서 변수값을 보관하는데 사용된다. 다음번 실례에서 함수 GetArg()는 실행평균을 계산한다. 그것은 이전에 평균한 수들의 총계와 수들의 개수를 기억한다. 새로운 수를 받아들일 때마다 total에 그 수를 더하고 count에 1을 더하며 total을 count로 나누어 새로운 평균을 돌려준다.

(실례 5-18) 정적변수

```
#include <iostream>
using namespace std;
float GetAvg(float);
int main()
{
  float data = 1, avg;
  while(data != 0)
  {
    cout << "수를 입력하시오: ";
    cin >> data;
    avg = GetAvg(data);
    cout << "새로운 평균은 " << avg << endl;
```

```
    }
    return 0;
}
float GetAvg(float newData)
{
    static float total = 0;
    static int count = 0;
    count++;
    total += newData;
    return total / count;
}
```

아래에 프로그람과의 대화가 있다.

```
수를 입력하시오: 10
새로운 평균은 10      ← total=10, count=1
수를 입력하시오: 20
새로운 평균은 15      ← total=30, count=2
수를 입력하시오: 30
새로운 평균은 20      ← total=60, count=3
```

GetAvg()에서 정적변수 total과 count는 GetAvg()로부터 돌아온후에도 함수가 다시 호출될 때까지 계속 존재한다.

1) 초기화

정적변수는 함수가 처음으로 호출될 때 한번만 초기화된다. 정적변수는 다음번 함수호출에서는 초기화되지 않는다.

2) 기억등급

조작체계의 구조를 알고있다면 자동변수가 탄창우에 보관되는 방법과 외부변수와 정적변수가 히프(heap)에 보관되는 방법을 알수 있다.

표 5-2에 자동, 정적자동, 외부변수의 수명, 보임성, 기타 특성을 설명하였다.

표 5-2 . 기억형태

	자동	정적자동	외부
보임성	함수	함수	파일
수명	함수	프로그람	프로그람
초기화	초기화되지 않는다	초기화된다	초기화된다
기억장소	탄창	히프	히프
목적	변수는 단일함수에 의해 사용된다	자동과 같은데 함수가 끝날 때 값을 보관한다.	여러 함수들이 변수를 사용한다.

제 8 절. 참고에 의한 귀환과 const함수인수

1. 참고에 의한 귀환

이제는 외부변수를 알고있으므로 C++의 특성을 시험할수 있다. 또한 참고에 의해 값을 넘기고 참고에 의해 값을 돌려줄수 있다. 무엇때문에 참고에 의해 값을 돌려주어

야 하는가는 명백하다. 그 리유는 우선 =기호의 왼변에 함수호출을 사용할수 있게 하려는데 있다. 이것은 특이한 개념이므로 실례를 들기로 하자.

(실례 5-19) 참고값의 돌려주기

```
#include <iostream>
using namespace std;
int x;
int& SetX();
int main()
{
   SetX() = 92;
   cout << "x= " << x << endl;
   return 0;
}
int& SetX()
{
   return x;
}
```

여기서 함수 SetX()는 참고형 즉 돌림값형 int&로서 선언된다.

```
int& SetX();
```

이 함수는 명령문

```
return x;
```

를 호출한다. 여기서 x는 외부변수로서 정의된다. 그러면 =기호의 왼변에서 이 함수를 호출할수 있다.

```
SetX() = 92;
```

따라서 함수로부터 돌아온 변수에 =기호의 오른변값이 대입된다. 즉 x에는 어떤 값이 주어진다. 프로그람의 출력은

```
x = 92
```

- 같기기호의 왼변에서 함수호출

값을 돌려주는 함수를 어떤 값으로 사용할수 있다. 즉

```
y = SquareRoot(x);
```

여기서는 함수자체가 값으로 취급되고 SquareRoot(x)값이 y에 대입된다. 또한 참고를 돌려주는 함수는 변수처럼 사용된다. 그것은 변수 즉 함수의 return명령문에 있는 변수의 별명을 돌려준다.

실례 4-19에서 함수 SetX()는 변수 x에로의 참고를 돌려준다. SetX()를 호출할 때 마치도 그것이 변수 x인것처럼 취급되므로 SetX()를 =기호의 왼변에서 사용할수 있다.

여기에 주의할 점이 두가지 있다. 하나는 참고에 의해 돌아오는 함수로부터 상수를 돌려줄수 없는것이다. SetX()에서

```
int& SetX()
{
   return 3;
}
```

라고 쓸수 없다.

이것을 실행하면 번역프로그람은 왼변값이 요구된다고 통보한다. 즉 =기호의 왼변에로 돌려줄수 있는것 즉 상수가 아니라 변수가 요구된다고 통보한다.

또 하나는 자동변수에로의 참고를 돌려줄수 없는것이다.

```
int& SetX()
{
  int x=3;
  return x;   // 오유
}
```

여기서는 함수의 자동변수가 함수로부터 돌아올 때 파괴되므로 더는 존재하지 않는데 자동변수에로의 참고를 돌려보내려고 한다. 문제는 =기호의 왼변에 함수호출을 사용하려고 하기때문에 발생한다.

수속적프로그람작성에서 이 방법은 그리 많이 쓰이지 않는다. 우의 실례와 같은 결과를 얻는 더 간단한 방법도 있다. 그러나 8장에서 참고에 의한 귀환이 반드시 필요한 기술이라는것을 알수 있다.

2. const함수인수

참고에 의한 인수넘기기가 호출측프로그람에 있는 변수를 변경하는데 함수를 사용하게 한다는것을 보았다. 그러나 참고에 의한 넘기기를 사용하는 다른 근거가 있다. 그 하나는 효과성이다. 함수인수로 사용되는 일부 변수들은 대단히 클수 있다. 실례로 큰 구조체를 들수 있다. 인수가 크면 참고에 의한 넘기기는 전체 변수를 넘기지 않고 그 주소만 넘기므로 더 효과있다.

효과성으로 하여 참고에 의해 인수를 넘기려고 하고 함수가 그것을 변경하려고 하지 않는다고 가정하자. 이때 함수가 변수를 수정하지 않는다고 담보해야 한다.

그러한 담보를 얻기 위하여 함수선언에서 변수에 const변경자를 적용할수 있다. 실례 5-20은 이것을 보여준다.

(실례 5-20) const함수인수

```
#include <iostream>
using namespace std;
void AFunc(int&, const int&);
int main()
{
  int alpha = 7;
  int beta = 11;
  AFunc(alpha, beta);
  return 0;
}
void AFunc(int& a, const int& b)
{
  a = 107;
```

```
        // b = 111; // 오유: 상수인수를 변경할수 없다
    }
```

여기서 AFunc()는 beta변수를 수정하지 않도록 담보한다. 또한 함수선언에서 b에 const변경자를 준다.

```
    void AFunc(int& a, const int& b)
```

그러면 Afunc()에서 beta를 변경하려는 시도는 번역프로그람오유로 된다.

C++의 설계원칙의 하나는 실행할 때에 오유가 나타나는것보다 번역프로그람이 오유를 찾게 하는것이다. const함수인수의 사용은 바로 그러한 실례이다. 함수에 const변수를 참고인수로서 넘기려고 한다면 함수선언에서 const로 선언해야 한다. 함수가 원시변수를 어떤 방법으로도 수정할수 없으므로 값에 의하여 const인수를 넘기는것은 문제없다. 많은 서고함수들에서는 이와 비슷한 방법으로 함수인수를 사용한다.

요 약

함수는 코드블로크에 이름을 주고 프로그람의 다른 부분으로부터 그것을 실행하게 함으로써 프로그람을 조직화하기 위한 방도를 제공하며 프로그람의 크기를 줄이게 한다. 함수선언은 함수형태를 지정하고 함수호출은 함수에로 조종을 넘기며 함수정의는 함수를 구성하는 명령문들을 포함한다. 함수선언자는 정의의 첫 행이다.

인수는 함수에서 값에 의해 넘길수 있다. 이때 함수는 인수의 사본을 가지고 작업한다. 또한 인수는 참고에 의해 함수에 넘길수 있다. 이때 함수는 호출측프로그람의 원시변수와 작업한다.

함수는 한개 값만 돌려줄수 있다. 보통 함수는 값을 돌려주지만 참고를 돌려줄수도 있다. 참고에 의한 귀환은 대입명령문의 왼변에서 함수호출을 사용할수 있게 한다. 인수와 돌림값은 기본자료형 혹은 구조체일수 있다.

재정의된 함수는 실제로 같은 이름을 가진 함수들의 묶음이다. 함수를 호출할 때 그것들중 어느 함수를 실행하는가 하는것은 호출에서 주어지는 인수들의 형과 개수에 의존한다.

inline함수는 원천파일안에서는 일반함수처럼 보이지만 호출측프로그람에 직접 함수코드를 삽입한다. inline함수는 고속으로 실행되지만 함수가 작지 않은 경우에는 일반함수보다 더 많은 기억기를 요구한다.

함수가 기정인수를 사용하면 그 함수호출에서는 선언에서 준 인수를 모두 포함할 필요가 없다. 함수에 의해 제공된 기정값은 인수의 생략에 쓰인다.

변수는 기억등급이라는 특성을 가진다. 가장 일반적인 기억등급은 자동이다. 자동기억등급의 변수는 그것이 정의된 함수가 실행중에 있을 때에만 존재하고 그 함수안에서만 볼수 있다. 외부변수는 프로그람의 수명기간 존재하고 전체 파일안에서 볼수 있다. 정적자동변수는 프로그람을 실행하는 전기간 존재하지만 자기 함수에서만 볼수

있다. 함수는 const변경자가 주어지는 인수를 변경하지 못한다. 호출측프로그람에서 이미 const로 정의된 변수는 const인수로 넘겨야 한다.

4장에서는 객체의 중요부분의 하나로서 자료집합인 구조체를 시험하였다. 이 장에서는 둘째 부분인 함수를 서술하였다. 이제는 이 두개 요소를 묶어서 객체를 창조할 준비가 되였다.

문 제

1. 함수의 가장 중요한 역할의 하나는
① 코드블로크에 이름을 주는것이다.
② 프로그람의 크기를 줄이는데 있다.
③ 인수를 받아들이고 돌림값을 돌려주는것이다.
④ 프로그람을 개념적인 단위들로 조직화하도록 도와준다.
어느것이 옳은가?
2. 함수자체를 무엇이라고 하는가?
3. 단어 foo를 표시하는 Foo라는 함수를 쓰시오.
4. 한개 명령문에 의한 함수서술을 무엇이라고 하는가?
5. 함수의 동작을 수행하는 명령문들은 무엇을 구성하는가?
6. 함수를 호출하는 프로그람명령문을 무엇이라고 하는가?
7. 함수정의의 첫 행을 무엇이라고 하는가?
8. 함수인수는
① 호출측프로그람으로부터 값을 받아들이는 함수안의 변수이다.
② 호출측프로그람의 값을 함수가 받아들이는것을 제한하는 방법이다.
③ 호출측프로그람이 함수에 넘기는 값이다.
④ 호출측프로그람에 함수가 돌려주는 값이다.
어느것이 옳은가?
9. 함수인수에 값이 넘어올 때 함수는 호출측프로그람안의 원시변수와 작업한다. 옳은가?
10. 함수선언에서 인수의 이름을 사용하는 목적은 무엇인가?
11. 다음의 어느것을 합법적으로 함수에 넘길수 있는가?
① 상수
② 변수
③ 구조체
④ 머리부파일
12. 함수선언에서 빈괄호의 의미는 무엇인가?

13. 함수로부터 몇개의 값을 돌려줄수 있는가?

14. =기호의 오른변에 함수호출식을 쓰는 방법으로 함수가 돌려주는 값을 다른 변수에 대입할수 있다. 옳은가?

15. 함수돌림값의 형을 어디에 지정하는가?

16. 아무것도 돌려주지 않는 함수는 어떤 돌림값형을 가지는가?

17. 아래에 다음의 함수가 있다.

```
int Time2(int a)
 {
  return (a * 2);
 }
```

이 함수호출에 필요한 모든것을 포함하는 main()프로그람을 쓰시오.

18. 인수를 참고에 의해 넘길 때

① 함수안에 변수가 창조되고 인수값을 보관한다.

② 함수는 인수값을 호출할수 없다.

③ 호출측프로그람에 림시변수가 창조되고 인수값을 보관한다.

④ 호출측프로그람안에 있는 변수의 원시값을 호출한다.

어느것이 옳은가?

19. 참고에 의하여 인수들을 넘기는 주되는 리유는 무엇인가?

20. 재정의된 함수는

① 같은 이름을 가지는 함수들의 묶음이다.

② 모두 같은 개수의 인수와 인수형을 가진다.

③ 프로그람작성자의 코드작성작업을 더 단순하게 해준다.

어느것이 옳은가?

21. Bar()라는 이름을 가지는 재정의된 함수의 선언을 두개 쓰시오. 둘다 돌림값은 int형이고 첫째 함수는 char형인수 하나, 둘째 함수는 char형인수 두개를 가진다. 만일 불가능하다면 무엇때문인가?

22. 일반적으로 inline함수는 속도와 기억기소비에서 일반함수와 어떻게 다른가?

23. float형인수를 하나 가지며 float형을 돌려주는 FooBar()라는 inline함수의 선언자를 쓰시오.

24. 기정인수는 다음의 한개값을 가질수 있다.

① 호출측프로그람에서 제공되는 값

② 함수에 의해 제공되는 값

③ 상수값

④ 변수값

어느것이 옳은가?

25. 두개 인수를 가지고 char형을 돌려주는 Blyth()라는 함수선언을 쓰시오. 제1인

수는 int, 제2인수는 float형, 그 기정값은 3.14159이다.

26. 기억등급은 변수의 어떤 특성과 관련되여있는가?

27. 어떤 함수들이 같은 파일에 있는 외부변수를 호출할수 있는가?

28. 자동변수를 어느 함수가 호출할수 있는가?

29. 정적자동변수는

① 여러개의 함수들에서 보이는 변수를 만드는데 사용된다.

② 한개 함수에만 보이는 변수를 만드는데 사용된다.

③ 함수를 실행하지 않을 때 기억기를 예약하기 위하여 사용된다.

④ 함수를 실행하지 않을 때 값을 얻는데 사용된다. 어느것이 옳은가?

30. 함수가 참고에 의해 값을 돌려줄 때 일반적으로 쓰이지 않는 어떤 위치에서 함수호출을 사용할수 있는가?

련습문제

1. 실례 2-6의 프로그람을 수정하시오. 그리고 같은 방법으로 원의 넓이를 구하는 함수 CircArea()를 정의하시오. 함수는 float형인수를 가지고 같은 형의 인수를 돌려준다. 사용자로부터 반경을 얻어서 CircArea()를 호출하고 결과를 표시하는 프로그람을 작성하시오.

2. n의 p제곱은 n을 p번 곱하는것과 같다. double형의 n값과 int형의 p값을 가지며 double형값을 결과로서 돌려주는 함수 Power()를 정의하시오. p의 기정값으로 2를 사용하여 이 인수를 생략하면 수 n을 두제곱하시오. main()함수에서는 사용자로부터 값을 얻어서 Power()함수를 시험하시오.

3. 두개의 int형인수를 참고로 넘기여 두 인수중 작은것을 0으로 설정하는 함수 ZeroSmaller()를 정의하고 main()에서 시험하시오.

4. 두개의 Distance값을 인수로 가지고 큰 값을 돌려주는 함수를 정의하시오. main()함수에서는 두개의 Distance값을 사용자로부터 받아들여 서로 비교하고 큰 값을 표시하시오.

5. int형인수 시, 분, 초를 가지고 시간을 초(long형)로 돌려주는 함수 HmsToSecs()를 정의하시오. 사용자로부터 12:59:59형식의 시, 분, 초값을 얻어서 HmsToSecs()함수를 호출하고 그것이 돌려준 초값을 표시하는 프로그람을 작성하시오.

6. 4장 련습 11 즉 두개의 Time구조체값들을 더하는 문제와 기능은 같지만 다음의 두개 함수를 사용하도록 프로그람을 수정하시오. 우선 TimesToSecs()는 Time구조체형의 값을 유일한 인수로 가지고 등가한 long형의 초값을 돌려준다. 둘째 함수 SecsToTimes()는 초로 된 시간(long형)을 유일한 인수로 가지고 Time구조체를 돌려준다.

7. 련습 2의 Power()함수와 이름이 같고 double, char, int, long, float형의 인수에 대하여 동작하는 함수들을 재정의하시오. main()함수에서 모든 형의 인수들에 대하여 재정의된 함수들을 시험하시오.

8. 호출측프로그람에서 넘어온 두개의 int값을 서로 교환하는 함수 Swap()를 정의하고 main()함수에서 시험하시오.

9. 련습 8에서 두개의 int형값대신 두개의 Time구조체값들을 서로 교환하는 함수 Swap()를 정의하시오.

10. 함수를 호출할 때 그 호출회수를 《나는 세번 호출되였습니다》라는 형식으로 표시하는 함수를 정의하시오. main()함수는 이 함수를 적어도 10번 호출하시오. 이 함수를 두가지 방법 즉 하나는 외부변수에 의하여 회수를 보관하는 방법, 다른 하나는 국부정적변수를 사용하는 방법으로 실현하시오. 어느 방법이 더 적합한가? 왜 자동변수를 사용할수 없는가?

11. 4장 련습 10의 GArea구조체에 기초한 프로그람을 작성하시오. 정보, 평단위 (15: 2000형식)로 두개의 토지면적을 사용자로부터 얻어서 그것들을 더하고 결과를 출력하시오. 다음의 세개 함수를 사용하시오. 첫째 함수는 사용자로부터 정보, 평을 얻어서 GArea구조체로서 값을 돌려주어야 한다. 둘째 함수는 GArea형의 두개의 인수를 가지고 인수들의 합을 같은 형으로 돌려주어야 한다. 셋째 함수는 GArea구조체를 인수로 가지고 그 값을 표시하여야 한다.

12. 4장 련습 12에서 매개 함수에서 인수로서 구조체를 사용하도록 4기능분수수산기프로그람을 수정하시오. 그 함수들은 FAdd(), FSub(), FMul(), FDiv()이며 Fraction 구조체형의 두개 인수를 가지고 같은 형의 값을 돌려준다.

제 6 장. 객체와 클라스

이미 자료요소들을 묶어두는 방법을 제공하는 구조체와 이름있는 실체들로 프로그람의 동작을 조직하는 함수들에 대하여 보았다. 이 장에서는 구조체와 함수를 하나로 묶어서 취급한다.

여기서는 간단한 클라스로부터 시작하여 복잡한 실체에 대하여 작업하는 클라스들을 소개한다. 먼저 클라스와 객체의 세부 즉 성원함수와 자료, private와 public, 구성자와 해체자에 대하여 고찰한다. 다음으로 현실세계에서의 객체란 무엇이고 객체를 언제 사용하는가에 대하여 설명한다.

이 장을 읽은 다음 1장에서 소개한 개념을 다시 참고하시오.

제 1 절. 클라스

첫 프로그람은 한개 클라스와 그 클라스의 두개 객체를 포함한다. 이것은 간단하지만 C++에서 클라스의 문법과 일반특성을 보여준다.

(실례 6-1) 간단한 객체

```
#include <iostream>
using namespace std;
class SmallObj
{
private:
   int someData;
public:
   void SetData(int d)
      { someData = d; }
   void ShowData()
      { cout << "자료=" << someData << endl; }
};
int main()
{
   SmallObj s1, s2;
   s1.SetData(1066);
   s2.SetData(1776);
   s1.ShowData();
   s2.ShowData();
   return 0;
}
```

이 프로그람에서 선언한 클라스 SmallObj는 한개의 자료항목과 두개의 성원함수를 가진다. 두개의 성원함수는 클라스밖에서 자료항목에 대한 유일한 호출을 제공한다. 첫째 성원함수는 자료항목에 값을 설정하고 둘째 함수는 그 값을 표시한다.

하나의 실체안에 자료와 함수들을 배치하는것은 객체지향프로그람작성법의 중요한 원칙이다. 이것을 그림 6-1에 보여주었다.

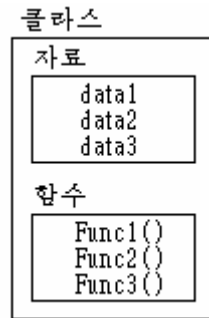



그림 6-1. 자료와 함수를 포함하는 클라스

1. 클라스와 객체

변수가 자료형을 가지는것처럼 객체도 클라스를 가진다.

객체를 어떤 클라스의 실례(instance)라고도 한다. 실례 6-1에서 클라스 SmallObj는 프로그람의 첫 부분에서 선언된다. 다음에 main()에서 두개의 객체 s1과 s2를 클라스의 실례로 정의한다.

두개의 객체는 각각 값을 설정하고 그 값을 표시한다. 여기에 프로그람의 출력이 있다.

```
자료=1066        <- 객체 s1이 이것을 표시한다.
자료=1776        <- 객체 s2이 이것을 표시한다.
```

프로그람의 첫 부분 즉 클라스 SmallObj의 선언을 고찰하자.

2. 클라스선언

여기에 클라스 SmallObj의 선언(지정자)이 있다.

```
class SmallObj      //클라스선언
{
  private:
      int someData;
  public:
      void SetData(int d)
      { someData=d;}
      void ShowData( )
      { cout << "자료=" << someData << endl; }
};
```

선언은 예약어 class로 시작되고 그 뒤에 클라스이름 SmallObj이 있다. 구조체처럼 클라스의 본체는 괄호안에 놓이고 반두점으로 끝난다.(반두점을 잊어서는 안된다. 자료는 구조체처럼 구성하고 클라스선언은 반두점으로 끝나며 조종은 함수처럼 구성하고 순환은 없다.)

- private와 public

클라스본체에는 두개의 다른 예약어 private와 public가 있다. 그 목적은 무엇인가?

객체지향프로그람작성법의 중요한 특징은 자료은페이다. 자료은페는 자료가 클라스안에 은페되므로 클라스밖의 함수들에 의해 호출할수 없다는것을 의미한다. 자료를 은페하는 원시기구는 클라스안에서 그것을 private로 하는것이다. 비공개자료 혹은 비공개함수들은 클라스안에서만 호출할수 있다. 공개자료 혹은 공개함수는 클라스밖에서 호출할수 있다. 그림 6-2에 그것을 보여주었다.

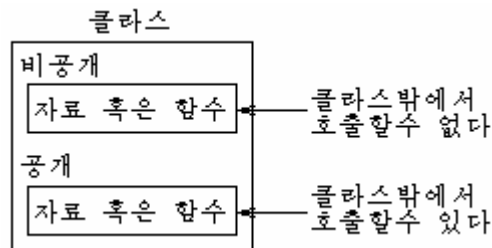



그림 6-2. private와 public

- 자료은페

콤퓨터자료기지를 보호하는데 사용하는 자료보호기술과 자료은페를 혼돈해서는 안된다. 자료보호를 제공하기 위해서는 실례로 사용자가 자료기지에로의 호출을 얻기전에 암호를 입력할것을 요구한다. 암호는 자료를 교환하도록 허용된 사용자를 구별하는데 쓰인다.

다른 한편 자료은페는 자료를 호출할 필요가 없는 프로그람부분들로부터 객체의 자료은페를 말한다. 더 정확히 말하면 어떤 클라스의 자료는 다른 클라스로부터 은페된다. 자료은페는 프로그람작성자들이 정당한 오유로부터 보호할 목적으로 설계된다.

프로그람작성자들은 비공개자료를 호출하기 위한 방법을 창안해내려고 하지만 우연적으로 그렇게 하기는 힘들다.

- 클라스자료

SmallObj클라스는 한개의 자료항목 즉 int형의 someData를 포함한다. 클라스안의 자료항목을 자료성원 또는 성원자료라고 한다. 구조체에 임의의 개수의 자료항목이 있을수 있는것처럼 클라스안에도 임의의 개수의 자료성원들이 있을수 있다. 자료성원 someData는 예약어 private뒤에 놓여있으므로 클라스안에서만 호출할수 있고 밖에서는 호출할수 없다.

- 성원함수

성원함수는 클라스안에 포함되는 함수이다. Smalltalk와 같은 객체지향언어에서는 성원함수를 메쏘드(method)라고 한다. 실례 6-1에는 두개의 성원함수 SetData()와 ShowData()가 있다. 함수들의 본체는 한 행에서 괄호안에 쓴다. 또한 함수정의의 일

반형식을 사용할수 있다.

```
void  SetData(int d)
{
    someData = d;
}
```

와

```
void ShowData()
{
    cout << "자료=" << someData << endl;
}
```

그러나 성원함수가 작을 때 이 방법으로 함수정의를 압축하면 공간을 절약할수 있다. SetData()와 ShowData()는 예약어 public뒤에 놓이므로 클라스의 밖에서 호출할수 있다. 그림 6-3은 클라스선언의 문법을 보여준다.

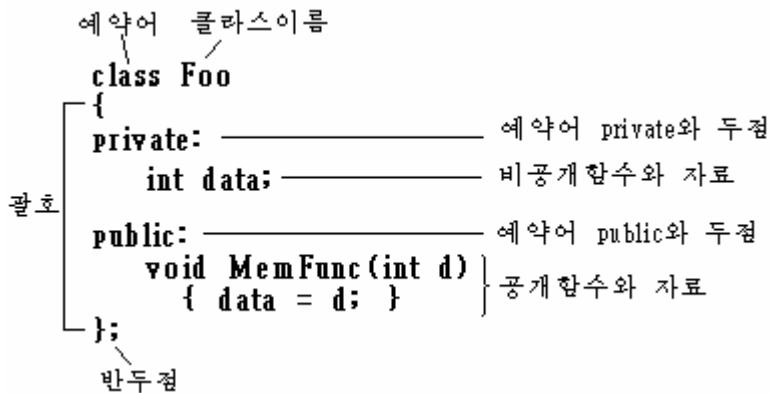



그림 6-3. 클라스지정자문법

- 함수는 공개, 자료는 비공개

보통 클라스의 자료는 비공개이고 함수는 공개이다. 자료는 은폐되므로 우연적인 조작으로부터 안전하고 자료에 대하여 조작하는 함수는 공개이므로 클라스의 밖에서 호출할수 있다. 그러나 반드시 자료를 비공개로 하고 함수를 공개로 해야 한다는 규칙은 없으며 어떤 경우에는 비공개함수와 공개자료를 사용해야 할 필요도 제기된다.

- 클라스선언안의 성원함수

SmallObj클라스의 성원함수들은 클라스에서 아주 보편적인 조작 즉 클라스안의 자료설정과 클라스에 보관된 자료의 얻기를 수행한다.

SetData()함수는 한개의 값을 파라메터로서 받아들이고 someData변수를 그 값으로 설정한다. ShowData()함수는 someData에 보관된 값을 표시한다. 성원함수 SetData()와 ShowData()를 정의하는 코드는 클라스선언안에 들어있다.

함수의 정의는 함수코드용으로 기억기가 설정된다는것을 의미하지 않는다. 함수용 기억기는 클라스의 객체가 창조될 때까지 설정되지 않는다. 클라스안에서 정의된 성원함수는 기정적으로 inline함수로서 창조된다. 일반적으로 클라스밖에서 정의된 함수는

inline함수가 아니다.

3. 클라스의 사용

그러면 클라스를 선언한 다음 main()에서 그것을 사용하는 방법을 고찰해보자. 즉 객체를 정의하는 방법과 성원함수를 호출하는 방법을 고찰한다.

- 객체의 정의

main()의 첫 명령문 SmallObj s1,s2;는 SmallObj클라스의 두개 객체 s1, s2을 정의한다. 이때 SmallObj클라스의 선언은 객체를 창조하지 않으며 객체가 창조될 때 그 모양을 서술한다. 마치도 구조체선언이 구조체변수를 창조하지 않고 구조체의 구조만 보여주는것과 같다. 실제로 프로그람에서 사용하는 객체를 창조하는것은 바로 정의이다. 객체의 정의는 임의의 자료형의 변수를 정의하는것과 같다. 이때 객체용의 공간이 기억기에 할당된다.

이처럼 객체의 정의는 객체의 창조를 의미한다. 또한 객체의 실례작성(instantiation)이라고도 한다. 실례작성이란 클라스의 실례를 창조하는것을 말한다. 객체는 클라스의 하나의 개별적인 실례이다.

객체를 실례변수(instance variable)라고도 부른다.

4. 성원함수의 호출

main()함수에서 다음의 두개 명령문은 성원함수 SetData()를 호출한다.

```
s1.SetData(1066);
s2.SetData(1776);
```

이 명령문들은 표준함수호출과 다르다. 여기서는 객체이름 s1과 s2를 함수이름과 점으로 련결한다. 이 문법은 특정한 객체와 련결되는 성원함수를 호출하는데 사용된다. SetData()는 SmallObj클라스의 성원함수이므로 항상 SmallObj클라스의 객체와 결합되여 호출되여야 한다. 이것은

```
SetData(1066);
```

과 의미가 다르다. 일반적으로 성원함수는 클라스가 아니라 객체에 대하여 동작할 때에만 호출할수 있다. 클라스의 성원함수는 그 클라스의 객체에 의해서만 호출될수 있다.

성원함수를 사용하려면 점(dot, period)연산자에 의하여 객체이름과 성원함수를 련결하여야 한다. 이 문법은 구조체성원을 참고할 때와 같지만 괄호는 자료항목을 참고하는것이 아니라 성원함수를 실행한다는것을 의미한다. 또한 점연산자를 클라스성원호출연산자(class member access operator)라고도 한다.

SetData()의 첫번째 호출

```
s1.SetData(1066);
```

은 s1객체의 SetData()성원함수를 호출한다. 이 함수는 s1객체의 변수 someData를

값 1066으로 설정한다. 두번째 호출

```
s2.setdata(1776);
```

은 s2객체의 변수 someData를 1776으로 설정한다. 그림 6-4는 someData변수들이 각이한 값을 가지는 객체라는것을 보여준다.

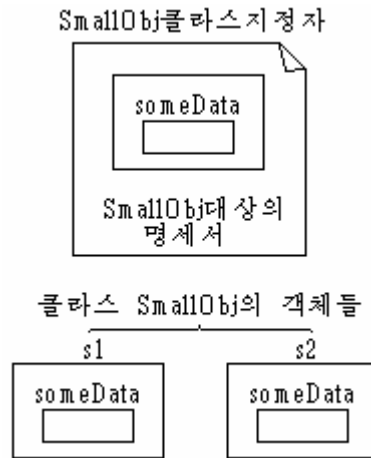



그림 6-4. 클라스 SmallObj의 두개 객체

다음과 같이 someData()함수를 두번 호출하면 두개의 객체가 자기 값을 표시한다.

```
s1.ShowData();
s2.ShowData();
```

- 통보문

일부 객체지향언어에서는 성원함수호출을 통보문으로 서술한다. 그러므로

```
s1.ShowData();
```

의 호출은 s1에 자료를 표시하라는 통보문을 보내는것과 같다.

통보문(message)이라는 단어는 C++에서 일반적인 용어가 아니지만 성원함수를 론의할 때 늘 생각해야 할 유용한 개념이다. 통보문에 대한 언급은 객체들이 서로 구별되는 실체이고 성원함수를 호출하는 방법으로 객체들이 서로 교제한다는것을 강조한다.

5. 물리적객체로서 C++객체

대부분의 프로그람작성환경에서 프로그람안의 객체들은 물리적객체 즉 사물현상을 표시한다. 이러한 상황은 프로그람과 현실세계사이의 대응관계의 뚜렷한 실례를 제공한다. 그러한 경우로서 두가지 즉 부분품객체와 직4각형 도형객체를 고찰하자.

- 부분품객체

마지막 실례에서 SmallObj클라스는 하나의 자료항목을 가지고있다. 더 명백한 클라스의 실례를 고찰해보자. 실례 4-1에서 보았던 부분품구조체 Part에 기초하여 클라스를 창조한다.

(실례 6-2) 객체로서의 기계부분품

```
#include <iostream>
using namespace std;
class Part
{
private:
   int modelNumber;
   int partNumber;
   float cost;
public:
   void SetPart(int mn, int pn, float c)
   {
      modelNumber = mn;
      partNumber = pn;
      cost = c;
   }
   void ShowPart()
   {
      cout << "형번호=" << modelNumber;
      cout << ", 부분품번호=" << partNumber;
      cout << ", 단가=" << cost << endl;
   }
};
int main()
{
   Part part1;
   part1.SetPart(6244, 373, 217.55F);
   part1.ShowPart();
   return 0;
}
```

이 실례는 클라스 Part의 기능을 보여준다. 실례 6-1과는 달리 세개의 자료항목 modelNumber, partNumber, cost를 가진다. 성원함수 SetPart()는 세개의 자료항목 모두에 값을 설정한다. 다른 성원함수 ShowPart()는 세개의 항목에 보관한 값을 표시한다.

이 실례에서는 Part형의 한개 객체 part1을 창조한다. 성원함수 SetPart()는 Part의 세개 자료항목에 값 6244, 373, 217.55를 설정한다. 그리고 성원함수 ShowPart()는 그 값을 표시한다. 출력은 다음과 같다.

```
형번호=6244, 부분품번호=373, 단가=217.55
```

이것은 실례 6-2보다 실천적인 실례이다. 부분품관리프로그람을 설계할 때 실례 6-2와 류사한 클라스의 객체를 실제로 창조할수 있다. 이것은 현실세계의 물리적객체 즉 부분품을 표시하는 C++객체의 실례이다.

- 직4각형객체

다음의 실례에서는 직4각형을 표시하는데 쓰이는 객체를 시험한다. 화상은 사람이 손으로 쥘수 있는 부분품과 같은 물리적객체는 아니지만 프로그람을 실행할 때 직4각

형과 같은것을 반드시 볼수 있다.

실례 6-3은 실례 5-6의 객체지향판이다. 이 프로그람은 여러가지 특성을 가지는 세개의 직4각형을 창조하고 면적을 표시한다.

(실례 6-3) 도형객체로서의 직4각형

```
#include <iostream>
using namespace std;
class Rectangle
{
private:
   int width;
   int height;
public:
   void Set(int w, int h)
   {
      width = w;
      height = h;
   }
   void Area()
   { cout << "면적=" << width * height << endl; }
};
int main()
{
   Rectangle r1;
   Rectangle r2;
   Rectangle r3;
   r1.Set(15, 7);
   r2.Set(41, 12);
   r3.Set(65, 18);
   r1.Area();
   r2.Area();
   r3.Area();
   return 0;
}
```

두개의 프로그람을 비교할수 있다. 실례 6-3에서 매개 직4각형은 실례 5-6처럼 구조체변수와 그것과 련관되지 않은 Area()함수에 의해서가 아니라 C++의 객체로 표시된다. 직4각형과 관련한 모든 속성과 함수는 클라스선언에 있다.

실례 6-3에서는 Area()함수외에 속성을 설정하는 하나의 인수를 가지는 함수 Set()를 요구한다.

6. 자료형으로서 C++객체

여기에 C++객체가 표시하는 다른 종류의 실체로서 사용자정의자료형의 변수가 있다. 4장에서 이미 언급한 거리를 표시하는데 객체를 사용한다.

(실례 6-4) 거리객체

```
#include <iostream>
using namespace std;
class Distance
{
private:
   int meters;
   float centies;
public:
   void SetDist(int me, float ce)
   { meters = me, centies = ce; }
   void GetDist()
   {
      cout << "\n메터를 입력하십시오:";
      cin >> meters;
      cout << "센치메터를 입력하십시오:";
      cin >> centies;
   }
   void ShowDist()
   { cout << meters << "m " << centies << "cm"; }
};
int main()
{
   Distance dist1, dist2;
   dist1.SetDist(11, 6.25);
   dist2.GetDist();
   cout << "\ndist1="; dist1.ShowDist();
   cout << "\ndist2="; dist2.ShowDist();
   cout << endl;
   return 0;
}
```

이 프로그람에서 Distance클라스는 두개의 자료항목으로서 메터와 센치메터를 표시한다.

이것은 4장에서 본 Distance구조체와 비슷하지만 여기서 Distance클라스는 세개의 성원함수 즉 인수들을 사용하여 메터와 센치메터를 설정하는 SetDist(), 사용자로부터 건반을 통하여 메터와 센치메터값을 얻는 GetDist(), 메터와 센치메터형식으로 거리를 표시하는 ShowDist()를 가지고있다.

Distance클라스의 객체의 값은 두가지 방법으로 설정할수 있다. main()에서는 Distance클라스의 두개 객체 dist1, dist2를 정의한다. 첫째 객체는 인수로서 11과 6.25를 가지고 성원함수 SetDist()를 통하여 값을 얻으며 둘째 객체는 사용자가 입력하는 값을 얻는다. 프로그람과의 실행결과는 다음과 같다.

```
메터를 입력하십시오: 10
센치메터를 입력하십시오: 4.75
dist1=10m 6.25cm
dist2=4m 75cm
```

제 2 절. 구성자

실례 6-4는 객체의 자료항목에 값을 주는데 성원함수를 사용하는 두가지 방법을 보여준다. 그러나 객체의 초기화는 성원함수를 따로 호출하지 않고 객체를 창조할 때 수행하는것이 관례로 되여있다. 자동초기화는 구성자라는 특수성원함수에 의해 수행된다. 구성자(constructor 또는 ctor)는 객체를 창조할 때 자동적으로 실행되는 성원함수이다.

1. 계수기프로그람

실례로 일반적인 프로그람요소로 사용할수 있는 객체들의 클라스를 창조하자. 계수기는 사물을 계수하는 변수이다. 즉 파일호출회수 또는 사용자가 Enter건을 누른 회수, 은행에 들어오는 손님의 수를 계수하는데 사용할수 있다. 사건이 발생할 때마다 계수기는 하나씩 증가된다. 또한 현재의 계수값을 얻기 위하여 계수기를 호출할수도 있다.

계수기가 프로그람에서 중요하고 여러개의 함수에서 호출될수 있어야 한다고 가정하자. C와 같은 수속형언어에서는 계수기를 외부변수로서 실현한다. 그러나 외부변수는 프로그람의 설계를 복잡하게 하고 잘못 변경될수도 있다.

실례 6-5는 성원함수를 통해서만 변경할수 있는 계수기변수를 제공한다.

(실례 6-5) 계수기변수를 표시하는 객체

```
#include <iostream>
using namespace std;
class Counter
{
private:
   unsigned int count;
public:
   Counter() : count(0)
      { /* 빈 본체 */ }
   void IncCount() { count++; }
   int GetCount() { return count; }
};
int main()
{
   Counter c1, c2;
   cout << "\nc1=" << c1.GetCount();
   cout << "\nc2=" << c2.GetCount();
   c1.IncCount();
   c2.IncCount();
   c2.IncCount();
   cout << "\nc1=" << c1.GetCount();
   cout << "\nc2=" << c2.GetCount();
```

```
    cout << endl;
    return 0;
}
```

Counter클라스는 unsigned int형의 한개 자료성원 count(항상 정의 옹근수)를 가진다. 세개의 성원함수 즉 구성자 Counter(), 계수기를 증가시키는 IncCount(), 현재 계수기값을 얻는 GetCount()가 있다.

1) 자동초기화

Counter형객체를 창조할 때 count를 0으로 초기화하여야 한다. 보통 계수는 0으로부터 시작된다. 그러기 위하여 SetCount()함수를 제공하여 인수값 0으로 호출하거나 ZeroCount()함수를 제공하여 count를 항상 0으로 설정할수 있다. 그러나 이러한 함수는 Counter객체를 창조할 때마다 실행해야 한다.

```
Counter c1;
c1.ZeroCount();
```

만일 프로그람작성자가 객체를 창조한 다음 초기화를 잊어버리는 경우에는 이것이 오유의 근원으로 된다. 특히 주어진 클라스의 객체들이 매우 많을 때에는 객체를 창조할 때 그 자체를 초기화하는것이 더 믿음직하고 관례적인 방법이다. Counter클라스에서는 Counter()구성자가 이것을 수행한다. 구성자함수는 Counter형객체를 새로 창조할 때 항상 자동적으로 호출된다. 따라서 main()에서 명령문

```
Counter c1, c2;
```

은 Counter형의 두개 객체를 창조한다. 매개 객체를 창조할 때 그 구성자 Counter()가 실행된다. 구성자함수는 count변수를 0으로 설정한다. 즉 이 명령문은 두개의 객체를 창조할뿐아니라 그것들의 count변수를 0으로 초기화한다.

- 클라스와 같은 이름

구성자함수는 다른 성원함수와 구별되는 특성을 가진다.

첫째로, 구성자는 클라스이름과 정확히 같아야 한다. 이것은 번역프로그람이 구성자를 식별하는 첫번째 방법이다.

둘째로, 구성자에는 돌림값을 쓰지 않는다. 무엇때문인가?

구성자는 체계에 의하여 자동적으로 호출되므로 값을 돌려줄 필요가 없다. 이것은 번역프로그람이 구성자를 식별하는 두번째 방법이다.

2) 초기화자목록

구성자가 수행하는 가장 중요한 일감의 하나는 자료성원의 초기화이다. Counter클라스에서 구성자는 count성원을 0으로 초기화해야 한다. 이것은 구성자함수의 본체에서 수행하는것으로 생각할수 있다. 즉

```
Counter() { count=0; }
```

그러나 이것은 좋은 수법이 아니고 자료성원을 초기화하는 다른 방법이 있다.

```
Counter() : count(0) { }
```

초기화는 성원함수선언자의 뒤와 함수본체의 앞에서 선언한다. 초기화앞에 두점을

삽입하고 성원자료뒤에 오는 괄호안에 값을 배치한다.

여러개의 성원을 초기화해야 할 때에는 반점으로 구분한다. 이것을 초기화자목록 (initializer list) 또는 성원초기화자목록이라고 한다.

```
SomeClass() : m1(7),m2(33),m2(4) ← 초기화자목록
{ }
```

구성자의 본체안에서 성원들을 초기화하지 않는 리유는 구성자를 실행하기 전에 초기화자목록에서 성원들에 초기값을 주어야 하기때문이다. 이것은 일부 경우에 아주 중요하다. 실례로 초기화자목록은 const성원자료와 참고를 초기화하는 유일한 방법이다.

구성자본체에서는 일반함수에서처럼 단순초기화보다 더 복잡한 동작을 서술한다.

3) 함수의 서술형식

이 실례에서는 함수들을 각각 두 행씩 차지하도록 서술하였다.

```
Counter()
{ count = 0; }
```

이것은 보통의 함수서술과 같다. 즉

```
Counter()
{
    count = 0;
}
```

프로그람의 main()부분에서는 Counter를 사용하여 두개의 계수기 c1과 c2을 창조하고 계수기들의 값을 0으로 초기화하며 값들을 표시하고 c1을 1번, c2을 2번 증가시킨 다음 계수기들의 값을 다시 표시한다. 출력은 다음과 같다.

```
c1=0
c2=0
c1=1
c2=2
```

구성자의 동작을 충분히 리해하기 위하여 구성자를 실행할 때 통보문을 출력하도록 할수 있다. 즉

```
Counter() : count (0)
{ cout << “나는 Counter입니다.\n” ; }
```

그러면 프로그람의 출력은 다음과 같아진다.

```
나는 Counter입니다.
나는 Counter입니다.
c1=0
c2=0
c1=1
c2=2
```

main()에서 명령문

```
Counter c1, c2;
```

을 실행할 때 구성자는 c1에 대하여 한번, c2에 대하여 한번씩 실행된다.

구성자는 아주 중요하다. C와 Basic, C++와 같은 언어의 사용자가 변수를 정의할

때에는 반드시 구성자가 기억할당과 초기화를 수행해야 한다. 가령 int형변수를 정의한다면 어디엔가 4byte의 기억기를 그 변수에 할당하는 구성자가 있어야 한다. 자체의 구성자를 쓸수 있다면 번역프로그람작성자의 일감을 덜어줄수 있다. 이것은 자체의 자료형을 창조하기 위한 첫 단계이다.

2. 도형 실례

실례 6-3의 프로그람을 Set()함수대신에 구성자를 사용하도록 다시 작성하자.

직4각형의 4개 속성의 초기화를 조종하려면 구성자에는 초기화목록에 두개의 인수와 두개의 항목이 있어야 한다. 여기에 실례 6-6이 있다.

(실례 6-6) 도형객체로서 직4각형

```
#include <iostream>
using namespace std;
class Rectangle
{
private:
   int width;
   int height;
public:
   Rectangle(int w, int h) : width(w), height(h) { }
   void Area() { cout << "면적=" << width * height << endl; }
};
int main()
{
   Rectangle r1(15, 7);
   Rectangle r2(41, 12);
   Rectangle r3(65, 18);
   r1.Area();
   r2.Area();
   r3.Area();
   return 0;
}
```

이 프로그람은 Set()가 구성자와 교체된것을 제외하면 실례 6-3과 비슷하다. 이것은 main()을 간단하게 한다. 객체마다 두개의 명령문 즉 객체를 창조하는 명령문과 객체의 속성을 설정하는 명령문대신에 한개 명령문에 의해 객체를 창조하고 속성을 설정한다.

3. 해체자

지금까지 객체를 창조할 때 자동적으로 호출되는 특수성원함수인 구성자를 보았다. 이번에는 객체가 해체될 때 자동적으로 호출되는 다른 함수를 고찰한다. 이러한 함수를 해체자(destructor, dtor)라고 한다. 해체자는 구성자와 같은 이름을 가지지만 앞에 물결표(~)가 있다. 즉

```
class Foo
{
private:
  int data;
public:
  Foo() : data(0) {}
  ~Foo() {}
};
```

구성자처럼 해체자는 돌림값을 가지지 않는다. 또한 객체를 해체하는데 목적이 있으므로 인수도 가지지 않는다.

해체자의 가장 일반적인 용도는 구성자에 의해 객체에 할당된 기억기를 해방하는 것이다.

제 3 절. 함수에서 객체의 사용

다음의 실례에서는 실례 6-4에 몇가지 기능을 추가한다. 또한 클라스의 새로운 특성 즉 구성자의 재정의와 클라스밖에서 성원함수의 정의, 함수인수로서 객체를 보여준다.

(실례 6-7) 구성자, 성원함수를 사용한 객체들의 더하기

```
#include <iostream>
using namespace std;
class Distance
{
private:
  int meters;
  float centies;
public:
  Distance() : meters(0), centies(0) { }
  Distance(int me, float ce) : meters(me), centies(ce) { }
  void GetDist()
  {
     cout << "\n메터를 입력하십시오:"; cin >> meters;
     cout << "센치메터를 입력하십시오:"; cin >> centies;
  }
  void ShowDist() { cout << meters << "m " << centies << "cm"; }
  void AddDist(Distance, Distance);
};
void Distance::AddDist(Distance d2, Distance d3)
{
  centies = d2.centies + d3.centies; meters = 0;
  if(centies >= 100.0)
  {
     centies -= 100.0; meters++;
  }
```

```
    meters += d2.meters + d3.meters;
  }
  int main()
  {
    Distance dist1, dist3;
    Distance dist2(11, 6.25);
    dist1.GetDist(); dist3.AddDist(dist1, dist2);
    cout << "\ndist1="; dist1.ShowDist();
    cout << "\ndist2="; dist2.ShowDist();
    cout << "\ndist3="; dist3.ShowDist(); cout << endl;
    return 0;
  }
```

이 실례에서는 거리 dist2에 초기값을 설정하고 거기에 사용자로부터 받아들인 거리 dist1을 더하여 거리의 합을 얻는다. 그리고 세개의 거리를 모두 표시한다.

```
메터를 입력하십시오:17
센치메터를 입력하십시오:5.75
dist1=17m 5.75cm
ndist2=11m 6.25cm
ndist3=28m 12cm
```

이 프로그람에 실현된 새 기능을 고찰하자.

1. 재정의된 구성자

Distance형변수가 창조될 때 그 변수에 값을 주어야 한다. 즉 다음과 같은 정의를 사용해야 한다.

```
Distance width(5, 6.25);
```

이것은 객체 width를 정의하고 동시에 메터를 5, 센치메터를 6.25로 초기화한다. 이렇게 하려면 구성자를 다음과 같이 써야 한다.

```
Distance(int me, float ce) : meters(me), centies(ce)
  {}
```

이 구성자는 성원자료 meters와 centies를 구성자에 인수로서 넘어오는 값으로 설정한다.

```
Distance dist1,dist3;
```

여기에는 구성자가 없으나 정의는 제대로 동작한다.

그러면 구성자가 없이 프로그람이 어떻게 작업하는가?

인수없는 구성자를 클라스에서 정의하지 않아도 번역프로그람에 의하여 프로그람에는 인수없는 구성자가 암시적으로 자동작성되고 그것이 객체를 창조한다. 인수없는 구성자를 기정구성자(default constructor)라고 한다. 객체가 구성자에 의해 자동적으로 창조되지 않는다면 구성자가 정의되지 않은 클라스의 객체들을 창조할수 없다.

그러므로 기정구성자에서 자료성원들을 초기화할 필요가 제기된다. 기정구성자가 초기화를 수행하게 할수 있지만 자료성원들에 주어야 할 값을 모른다. 만일 기정구성자에 주어야 하는 값을 알고있다면 구성자를 명백히 정의하여야 한다. 실례 6-7은 그

방법을 보여준다.

```
Distance():meters(0), conties(0.0) //기정구성자
{ } //함수본체가 비여있다. 아무것도 하지 않는다.
```

자료성원은 상수값으로 초기화된다. 즉 이 경우에 meters와 centies는 각각 옹근수값 0과 float값 0.0을 가진다. 이제는 인수없는 구성자에 의해 초기화된 객체를 사용할수 있고 객체들이 임의의 값이 아니라 거리가 없다는것(0m 0.0cm)을 확인할수 있다.

같은 이름을 가진 두개의 명시적인 구성자가 있으므로 구성자를 재정의한다고 말한다. 객체를 창조할 때 두개 구성자중 어느 구성자를 실행하는가 하는것은 정의에서 인수들을 사용하는 방법에 의존된다.

```
Distance   length ;          // 첫째 구성자호출
Distance   width(11,6.0)     // 둘째 구성자호출
```

2. 클라스밖에서 정의된 성원함수

지금까지 클라스선언안에서 정의된 성원함수들을 보았다. 그러나 실례 6-7에서 성원함수 AddDist()는 Distance클라스선언안에서 정의되지 않는다. 다만 클라스안에서 다음의 명령문

```
void AddDist(Distance, Distance);
```

으로 선언될뿐이다.

이것은 AddDist함수가 클라스의 성원이지만 클라스선언밖에서 정의된다는것을 번역프로그람에 알린다.

실례 6-7에서 AddDist()함수는 클라스선언의 후에 정의된다.

```
void Distance::AddDist(Distance d2, Distance d3)
{
   centies = d2.centies + d3.centies;
   meters = 0;
   if(centies >= 100.0)
   {
      centies -= 100.0;
      meters++;
   }
   meters += d2.meters + d3.meters;
}
```

이 정의에서 선언자는 새로운 문법을 사용한다. 함수이름 AddDist()는 클라스이름 Distance와 새 기호 즉 두개의 두점(::)뒤에 놓인다.

이 기호를 범위해결연산자(scope resolution operator)라고 한다. 이것은 함수와 련관된 클라스를 지정하는 하나의 방법이다. 이 경우에 Distance:: AddDist()는 《Distance의 AddDist()성원함수》를 의미한다. 그림 6-5는 그 사용법을 보여준다.

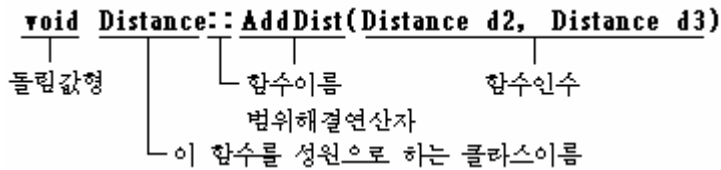



그림 6-5. 범위해결연산자

3. 인수로서의 객체

실례 6-7의 동작을 고찰하자. 거리 dist1과 dist3이 기정구성자에 의해 창조된다. 거리 dist2은 두개의 인수를 가지는 구성자에 의해 창조되고 인수들에 넘긴 값으로 초기화된다. 사용자로부터 값들을 얻는 성원함수 GetDist()를 호출하여 dist1의 값을 얻는다.

이제는 dist1과 dist2를 더하여 dist3을 얻어야 한다. main()에서 함수호출

```
dist3.AddDist(dist1,dist2);
```

이 이것을 수행한다. 더하려는 두개의 거리 dist1과 dist2은 AddDist()에 인수로서 공급된다. 객체인수의 문법은 int와 같은 기본자료형의 인수인 경우와 같다. 즉 객체의 이름은 인수로서 공급된다. 객체를 인수로서 사용하는 문법은 int와 같은 기본자료형의 인수인 경우와 같다. 즉 객체의 이름은 인수로서 공급된다. AddDist()가 Distance 클라스의 성원함수이므로 dist1.meters와 dist2.centies와 같은 이름을 사용하여 그 성원함수들에 인수로서 공급된 Distance클라스의 임의의 객체안의 비공개자료를 호출할 수 있다.

AddDist()를 통하여 성원함수에 대한 몇가지 중요한 사실들을 강조한다. 성원함수는 항상 그것을 호출한 객체와 점연산자에 의해 련결된다. 그러나 다른 객체들이 호출할수도 있다.

그러면 다음의 명령문에서 어느 객체가 AddDist()를 호출하는가?

```
dist3.AddDist(dist1,dist2);
```

성원함수를 호출한 객체는 dist3외에 dist1과 dist2를 호출해야 하므로 그것들을 인수로서 공급한다. 성원함수는 인수로 제공되지 않지만 그것을 성원으로 하는 객체에 대한 호출을 가진다. 그것은 이 명령문이 《dist3의 AddDist()성원함수의 실행》을 의미하기때문이다. 함수안에서 변수 meters와 centies를 참고할 때 그것은 dist3.meters와 dist3. centies를 참고한다.

AddDist()의 돌림값형이 void이므로 함수는 결과를 돌려주지 않는다. 결과는 dist3 객체에 자동적으로 보관된다. 그림 6-6은 두개의 거리 dist1과 dist2를 모두 더하여 dist3에 결과를 보관하는 과정을 보여준다.

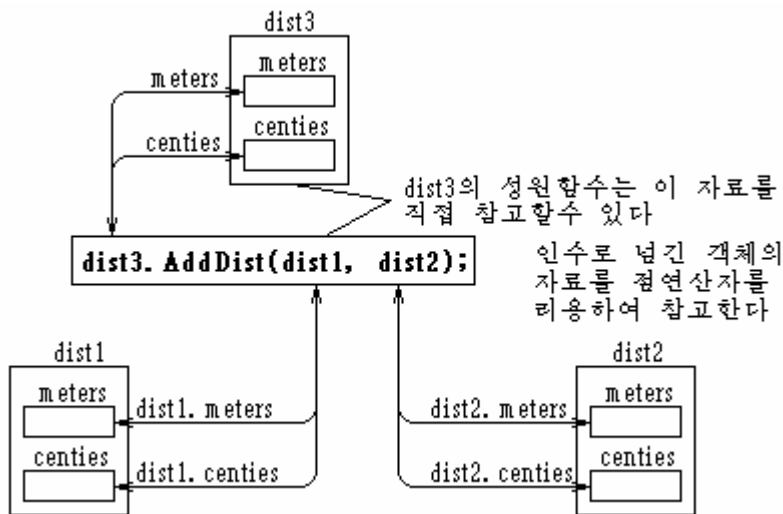



그림 6-6. 객체호출의 결과

요약하여 말하면 성원함수호출에 의해(그것이 정적함수가 아니라면) 개별적인 객체와 련결된다. 이 함수는 그 객체의 모든 성원들 즉 비공개 혹은 공개성원들을 성원이름만 가지고 직접 호출할수 있다. 또한 인수로서 넘어온 같은 클라스의 다른 객체들을 점연산자에 의하여 련결된 객체이름과 성원이름을 사용하여 간접호출할수 있다. (즉 dist1.meters와 dist2.centies)

4. 기정복사구성자

객체를 초기화하는 두가지 방법을 보았다. 인수없는 구성자는 자료성원들을 상수값으로 초기화할수 있으며 인수있는 구성자는 인수로서 넘어온 값들로 자료성원들을 초기화할수 있다.

객체를 초기화하는 다른 방법을 고찰하자. 어떤 객체를 같은 형의 다른 객체로 초기화할수 있다. 이것을 수행하는 특수한 구성자를 창조할 필요는 없으며 모든 클라스에 이미 만들어져있다. 그것이 기정복사구성자(default copy constructor)이다. 기정복사구성자는 인수가 구성자와 같은 클라스형의 객체인 1인수구성자이다. 실례 6-8은 기정복사구성자를 사용하는 방법을 보여준다.

(실례 6-8) 기정복사구성자를 사용한 객체의 초기화

```
#include <iostream>
using namespace std;
class Distance
{
private:
   int meters;
   float centies;
public:
   Distance() : meters(0), centies(0.0) { }
   Distance(int me, float ce) : meters(me), centies(ce) { }
```

```
    void GetDist()
    {
        cout << "\n메터를 입력하십시오:"; cin >> meters;
        cout << "센치메터를 입력하십시오:"; cin >> centies;
    }
    void ShowDist() { cout << meters << "m " << centies << "cm"; }
};
int main()
{
    Distance dist1(11, 6.25);
    Distance dist2(dist1);
    Distance dist3 = dist1;
    cout << "\ndist1="; dist1.ShowDist();
    cout << "\ndist2="; dist2.ShowDist();
    cout << "\ndist3="; dist3.ShowDist();
    cout << endl;
    return 0;
}
```

2인수구성자를 사용하여 dist1을 11m 6.25cm값으로 초기화한다. 그다음 Distance형의 두개 객체 즉 dist2와 dist3을 정의하여 dist1의 값으로 모두 초기화한다.

이것은 1인수구성자의 정의가 필요하다는것을 말해주지만 한 객체를 같은 형의 다른 객체로 초기화하는 특수한 경우이다. 이 정의에서는 모두 기정복사구성자를 사용한다. 객체 dist2은 명령문

```
Distance dist2(dist1);
```

에 의해 초기화된다.

이것은 Distance클라스의 기정복사구성자가 dist2에 대하여 dist1를 성원별로 복사하게 한다.

또한 다른 형식으로서 명령문

```
Distance dist3 = dist1;
```

에 의해 dist1이 dist3에 성원별로 복사된다.

이것은 대입명령문처럼 보이지만 대입이 아니다. 이 두가지 형식에서는 모두 기정복사구성자를 호출한다. 프로그람의 출력은 다음과 같다.

```
dist1=11m 6.25cm
dist1=11m 6.25cm
dist1=11m 6.25cm
```

이것은 dist2와 dist3객체들이 같은 값 dist1로 초기화된다는것을 보여준다.

5. 함수로부터 객체의 돌려주기

실례 6-8에서 함수에 객체를 인수로 넘기는 방법을 보았다. 그러면 객체를 돌려주는 함수의 실례를 보기로 하자.

(실례 6-9) 객체를 돌려주는 함수

```
#include <iostream>
using namespace std;
class Distance
{
private:
  int meters; float centies;
public:
  Distance() : meters(0), centies(0.0) { }
  Distance(int me, float ce) : meters(me), centies(ce) { }
  void GetDist() {
     cout << "\n메터를 입력하십시오:"; cin >> meters;
     cout << "센치메터를 입력하십시오:"; cin >> centies;
  }
  void ShowDist() { cout << meters << "m " << centies << "cm"; }
  Distance AddDist(Distance);
};
Distance Distance::AddDist(Distance d2)
{
  Distance temp;
  temp.centies = centies + d2.centies;
  temp.meters = 0;
  if(temp.centies >= 100.0) { temp.centies -= 100.0; temp.meters++; }
  temp.meters += meters + d2.meters;
  return temp;
}
int main()
{
  Distance dist1, dist3;
  Distance dist2(11, 6.25);
  dist1.GetDist();
  dist3 = dist1.AddDist(dist2);
  cout << "\ndist1="; dist1.ShowDist();
  cout << "\ndist2="; dist2.ShowDist();
  cout << "\ndist3="; dist3.ShowDist(); cout << endl;
  return 0;
}
```

실례 6-9는 실례 6-7과 비슷하지만 함수들이 객체들과 작업하는 중요한 방법을 보여준다.

- 인수와 객체

실례 6-7에서는 두개의 거리를 AddDist()에 인수로서 넘기며 결과는 AddDist()가 성원인 객체 즉 dist3에 보관된다. 실례 6-9에서는 거리 dist2를 AddDist()에 인수로서 넘긴다. 그것은 AddDist()가 성원으로 되는 객체 dist1에 더해지고 결과가 함수로부터 돌아온다. main()에서는 결과가 dist3에 대입된다. 즉

```
dist3=dist1.AddDist(dist2);
```

실례 6-7의 대응하는 명령문과 효과가 같고 대입연산자가 자연스럽게 사용되고있

다. 8장에서는 산수연산자 +를 더 명백한 식 dist3 = dist1 + dist2을 얻는데 사용하는 방법을 고찰한다.

여기에 실례 6-9의 AddDist()함수가 있다.

```
Distance Distance::AddDist(Distance d2)
{
  Distance temp;
  temp.centies = centies + d2.centies;
  temp.meters = 0;
  if(temp.centies >= 100.0) { temp.centies -= 100.0; temp.meters++; }
  temp.meters += meters + d2.meters;
  return temp;
}
```

이것을 실례 6-7에 있는 같은 함수와 비교하자. 알수 있는것처럼 일정한 차이가 있다. 실례 6-9에서는 Distance클라스의 림시객체가 창조된다. 림시객체는 호출측프로그람에로 돌아갈 때까지 두개 거리를 더한 합을 보관한다. 첫째 거리는 AddDist()가 성원인 객체 dist1로서 그 성원자료는 함수안에서 meters와 centies로서 호출된다. 둘째 거리는 인수로서 넘어온 객체 dist2이고 성원자료는 d2.meters와 d2.centies로서 호출된다. 결과는 temp에 보관되고 temp.meters와 temp.centies에 의해 호출된다. temp객체는 그다음 명령문 return temp;에 의하여 함수로부터 돌아오고 main()의 명령문이 그것을 dist3에 대입한다. 여기서 dist1은 변경되지 않고 단순히 AddDist()에 자료를 제공한다. 그림 6-7은 그 과정을 보여준다.

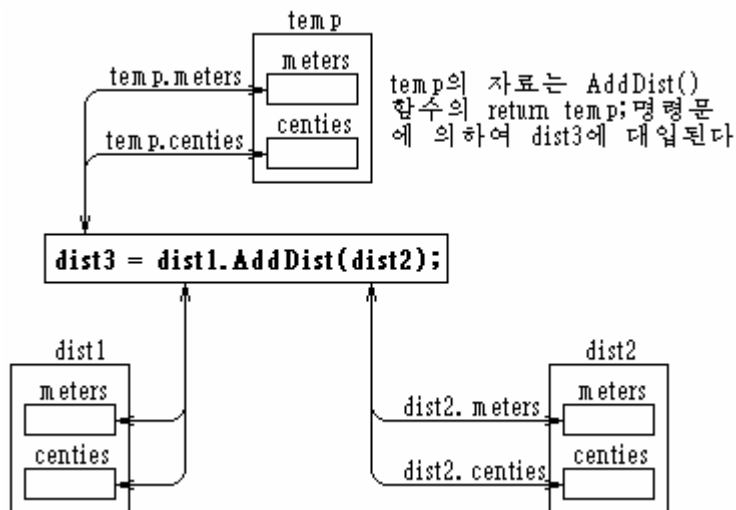



그림 6-7. 림시객체로서 돌려준 결과

- 카드유희의 실례

현실세계를 모형화하는 객체들의 실례로서 실례 4-6을, 객체를 사용하도록 변경한 실례 6-10을 고찰해보자. 여기서는 새 개념을 받아들이지 않지만 지금까지 론의한 프로그람작성방법을 거의 모두 사용한다. 실례 4-6과 같이 실례 6-10은 고정값을 가지

는 세개의 카드를 창조하고 사용자에게 그 위치를 알아맞추게 한다. 실례 6-10에서는 매 카드가 클라스 Card의 객체이다.

(실례 6-10) 객체로서의 카드

```
#include <iostream>
using namespace std;
enum Suit { clubs, diamonds, hearts, spades };
const int jack = 11; const int queen = 12;
const int king = 13; const int ace = 14;
class Card
{
private:
  int number;
  Suit suit;
public:
  Card() {}
  Card(int n, Suit s) : number(n), suit(s) {}
  void Display();
  bool IsEqual(Card);
};
void Card::Display()
{
  switch(suit) {
  case clubs: cout << "클라브 "; break;
  case diamonds: cout << "다야몬드 "; break;
  case hearts: cout << "하트 "; break;
  case spades: cout << "스페이드 "; break;
  }
  if(number >= 2 && number <= 10)
    cout << number;
  else {
    switch(number) {
    case jack: cout << "쟈크"; break;
    case queen: cout << "큐인"; break;
    case king: cout << "킹"; break;
    case ace: cout << "에이스"; break;
    }
  }
  cout << endl;
}
bool Card::IsEqual(Card c2)
{
  return (number == c2.number && suit == c2.suit) ? true : false;
}
int main()
{
  Card temp, chosen, prize;
  int position;
```

```
    Card card1(7, clubs);
    cout << "card1는 "; card1.Display();
    Card card2(jack, hearts);
    cout << "card2는 "; card2.Display();
    Card card3(ace, spades);
    cout << "card3는 "; card3.Display();
    prize = card3;
    cout << "card1과 card3을 교체합니다.\n";
    temp = card3; card3 = card1; card1 = temp;
    cout << "card2와 card3을 교체합니다.\n";
    temp = card3; card3 = card2; card2 = temp;
    cout << "card1과 card2를 교체합니다.\n";
    temp = card2; card2 = card1; card1 = temp;
    prize.Display(); cout << "의 위치(1,2,혹은 3)가 어디입니까? ";
    cin >> position;
    switch(position) {
    case 1: chosen = card1; break;
    case 2: chosen = card2; break;
    case 3: chosen = card3; break;
    }
    if(chosen.IsEqual(prize))
       cout << "옳습니다!\n";
    else
       cout << "틀렸습니다.\n";
    cout << "선택한것은 "; chosen.Display(); cout << "입니다.\n";
    return 0;
}
```

클라스 Card에는 두개의 구성자가 있다. 첫째 구성자는 인수를 가지지 않으며 main()에서 카드 temp, chosen, prize를 초기화하지 않고 창조하는데 쓰인다. 둘째 구성자는 두개의 인수를 가지며 card1, card2, card3을 창조하고 특정값으로 초기화하는데 사용된다. Card에는 구성자외에 두개의 다른 성원함수들이 있고 클라스밖에서 정의되였다.

Display()함수는 인수를 가지지 않고 그것을 성원으로 하는 카드객체의 number와 suit자료항목을 표시한다.

main()안의 명령문

```
chosen.Display();
```

는 사용자가 선택한 카드를 표시한다.

IsEqual()함수는 어떤 카드가 인수로서 제공한 카드와 같은가를 비교한다. 그것이 성원인 카드와 인수로서 공급된 카드를 비교하는데 조건연산자를 사용한다. 또한 if～else명령문을 사용하여 IsEqual()함수를 다음과 같이 쓸수도 있지만 조건연산자가 더 합리적이다.

```
if (number == c2.number && suit == c2.suit)
   return true;
```

```
else
   return false;
```

IsEqual()은 두개 카드를 비교해야 하므로 인수로서 c2을 호출한다. 첫째 자료는 IsEqual()이 성원으로 되여있는 객체이다. main()에서 식

```
if (chosen.isEqual(prize))
```

은 카드 chosen을 카드 prize와 비교한다.

여기에 사용자가 카드를 잘못 알아맞추는 출력이 있다.

```
card1는 클라브 7
card2는 하트 쟈크
card3는 스페이드 에이스
card1과 card3을 교체합니다.
card2와 card3을 교체합니다.
card1과 card2를 교체합니다.
스페이드 에이스의 위치(1, 2, 혹은 3)가 어디입니까? 1
틀렸습니다.
선택한것은 클라브 7입니다.
```

제 4 절. 구조체와 클라스, 클라스와 객체

1. 구조체와 클라스

지금까지의 실례들에서는 자료를 묶은 방법으로서 구조체, 자료와 함수들을 하나로 묶는 방법으로서 클라스를 고찰하였다. 사실상 클라스를 사용하는것과 꼭같은 방법으로 구조체를 사용할수 있다. 클라스와 구조체사이의 유일한 형식적인 차이는 호출지정자를 생략했을 때 클라스안에 있는 성원들은 기정으로 비공개이지만 구조체에서는 기정으로 공개라는데 있다.

여기에 클라스에서 사용한 형식이 있다.

```
class Foo
{
private:
   int data1;
public:
   void Func();
};
```

클라스에서는 private가 기정호출지정자이므로 생략할수 있다. 즉 다음과 같이 쓸수 있다.

```
class Foo
{
   int data1;
public:
   void Func();
};
```

여전히 data1은 비공개이다. 일부 프로그람작성자들은 이런 형식을 더 좋아한다. 그러나 예약어 private를 포함하여 명백히 하는것이 좋다.

이와 같은 방법으로 구조체를 사용하려면 private성원앞에 public성원을 넣어야 한다.

```
struct Foo
{
    void Func();
private:
    int data1;
};
```

구조체에서는 public가 기정호출지정자이다. 그러나 프로그람작성자들은 대체로 이러한 방법으로 구조체를 사용하지 않는다. 구조체는 오직 자료를 묶는데 사용되고 클라스는 자료와 함수들을 모두 묶는데 사용된다.

2. 클라스와 객체, 기억기

한 클라스로부터 창조된 매개 객체에는 클라스의 자료와 성원함수들의 서로 다른 사본이 포함되여있다. 이것은 객체가 클라스선언에 기초하여 설계되는 하나의 실례라는것을 강조한다.

실제로 매개 객체는 자기의 고유한 자료항목을 가지고있다. 다른 한편 주어진 클라스의 모든 객체는 같은 성원함수들을 가진다. 성원함수는 그것이 클라스선언에서 정의될 때 기억기안에 한번만 창조되여 배치된다.

클라스의 객체를 창조할 때마다 클라스의 성원함수들을 모두 반복하여 만들 필요는 없다. 그것은 매개 객체의 함수들이 같기때문이다. 그러나 자료항목은 각이한 값을 가지므로 객체마다 제각기 자료항목의 실례를 가지고있어야 한다. 그러므로 자료는 매개 객체를 정의할 때 기억기에 배치되고 객체마다 자기의 자료모임이 있다. 그림 6-8은 이것을 보여준다.

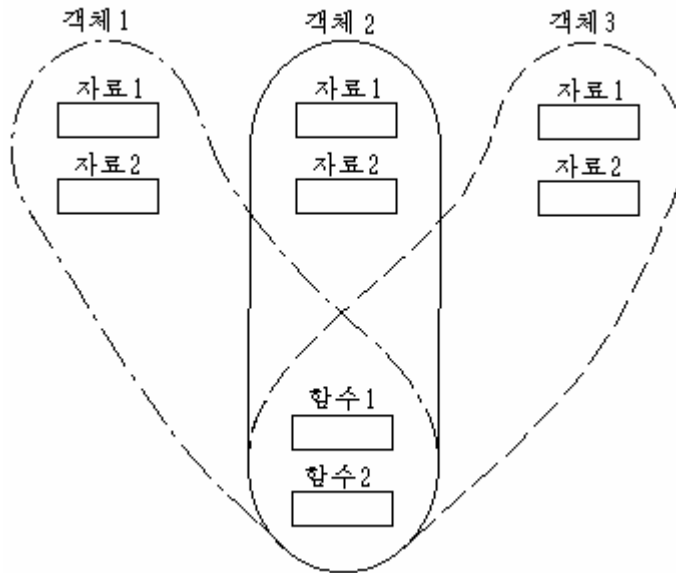



그림 6-8. 객체와 자료, 함수, 기억기

실례 6-1에서는 SmallObj형의 객체가 두개 있으므로 기억기에 someData의 실체가 두개 있다. 그러나 SetData()와 ShowData()함수의 실체는 각각 하나이다. 이 함수들은 그 클라스의 모든 객체들에 의하여 공유된다. 오직 한개의 함수가 한번에 실행되므로(적어도 단일스레드체계에서) 모순이 없다.

대부분의 경우에 전체 클라스에 대하여 성원함수가 오직 하나씩 있다는것을 알 필요는 없다. 객체자체의 자료와 성원함수들을 모두 포함하는 매개 객체에 대한 표상을 가지는것은 간단하다. 그러나 실행프로그람의 크기를 평가하는 경우에는 이와 같은 원리를 알아야 한다.

제 5 절. 정적클라스자료

매개 객체가 자기 자료를 가진다는것을 언급하였다. 클라스의 자료항목을 static로 선언하면 존재하는 객체의 수에 관계없이 전체 클라스에는 오직 한개의 자료항목이 창조된다. 정적자료항목은 클라스의 모든 객체들이 공통의 정보항목을 공유해야 할 때 효과가 있다. static로 선언된 성원변수는 정적변수와 같은 특성을 가진다. 즉 정적성원변수는 클라스안에서만 볼수 있지만 그 수명은 프로그람을 실행하는 전기간이다. 클라스의 항목이 더는 없어도 정적변수는 계속 존재한다. 일반정적변수는 함수에로의 호출들사이에서 정보를 보관하는데 사용되지만 정적클라스성원자료는 클라스의 객체들사이에서 정보를 공유하는데 사용된다.

1. 정적클라스자료의 사용

그러면 왜 정적성원자료를 사용하는가?

실례로 프로그람에 클라스의 객체들이 몇개 있는가를 알아야 한다고 하자.

경주경기를 례를 들면 경주용자동차는 경주로에 다른 차가 몇대 있는가를 알아야 할 필요가 있다. 이 경우에 정적변수 count를 그 클라스의 성원으로서 포함한다.

모든 객체들은 이 변수에 대한 호출을 가진다. 정적성원변수는 모든 객체에 대하여 같은 변수이며 모든 객체는 같은 count를 볼수 있다.

2. 정적클라스자료의 실례

간단한 정적자료성원을 보여주는 실례 6-11이 있다.

(실례 6-11) 정적클라스자료

```
#include <iostream>
using namespace std;
class AClass
{
private:
   static int count;
public:
   AClass() { count++; }
   int GetCount() { return count; }
};
int AClass::count = 0;
int main()
{
   AClass a1, a2, a3;
   cout << "count=" << a1.GetCount() << endl;
   cout << "count=" << a2.GetCount() << endl;
   cout << "count=" << a3.GetCount() << endl;
   return 0;
}
```

실례에서 클라스 Foo는 한개 자료항목 count를 가지는데 이것은 static int형이다. 클라스의 구성자는 count를 증가시킨다. main()에서는 Foo클라스의 세개 객체를 정의한다. 구성자가 세번 호출되므로 count는 세번 증가한다. 다른 성원함수 GetCount()는 count의 값을 돌려준다. 세개의 객체는 이 함수를 호출하고 매번 같은 값을 출력한다. 출력은 다음과 같다.

```
count is 3 ← 정적으로
count is 3
count is 3
```

만일 count를 정적변수대신 자동변수로서 선언한다면 매개 구성자는 자기의 count 비공개부분을 한번 증가시키며 따라서 출력은 다음과 같아진다.

```
count is 1
```

count is 1 ←자동자료
count is 1

정적클라스변수들은 자주 쓰이지 않지만 많은 경우에 중요한 역할을 한다. 그림 6-9는 정적변수와 자동변수를 대비적으로 보여준다.

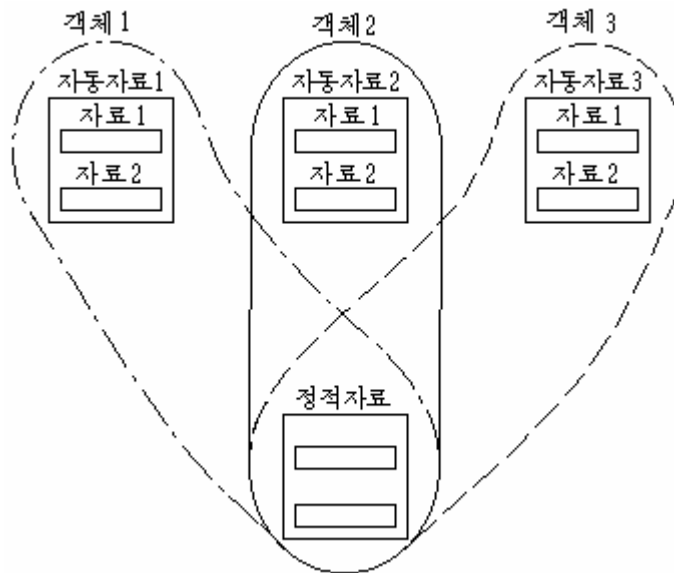



그림 6-9. 정적자동성원변수

3. 선언과 정의의 분리

정적성원자료는 비일반형식을 요구한다. 일반변수는 같은 명령문에서 선언되고(번역프로그람은 그것들의 형과 이름에 대하여 알린다.) 정의된다.(번역프로그람은 변수를 보관하는 기억기를 설정한다.)

다른 한편 정적성원자료는 두개의 개별적인 명령문을 요구한다. 변수의 선언은 클라스선언안에 있지만 그 변수는 실제로 외부변수와 같은 방법으로 클라스밖에서 정의된다.

그러면 왜 이렇게 두개 부분을 사용하는가?

정적성원자료가 클라스선언안에 정의되여있으면 클라스선언이 오직 설계도이고 어떤 기억기도 설정하지 않는다는 사실에 위반된다. 클라스밖에 정적성원자료의 정의를 배치함으로써 그 자료용의 기억공간이 프로그람이 실행을 시작하기 전에 오직 한번만 할당되고 전체 클라스에 의하여 한개의 정적성원변수가 호출된다는것, 매개 객체는 일반성원자료와 달리 그 변수의 자체의 판을 가지지 않는다는것을 강조하게 한다. 따라서 정적성원변수는 대역변수와 더 비슷하다.

정적자료를 잘못 조종하기 쉽고 번역프로그람은 그러한 오유를 통보하지 않는다. 정적변수를 선언하고 정의를 잊어버리면 번역프로그람이 경고하지 않고 련결할 때에

련결프로그람이 선언하지 않은 외부변수를 참고하려고 한다고 경고한다. 이러한 현상은 정의를 포함하였지만 클라스이름(실례 6-11에서 Foo::)을 잊었을 때에도 발생한다.

제 6 절. const와 클라스

일반변수를 그에 대한 변경으로부터 방지하는데 const를 사용한 실례를 여러개 보았으며 5장에서 참고에 의하여 함수에 넘긴 변수를 수정하지 않도록 하기 위해 함수 인수들과 사용할수 있다는것을 보았다. 클라스를 배웠으므로 const를 클라스에서 사용하는 방법 즉 성원함수, 성원함수의 인수 및 객체들에 사용하는 방법을 고찰하자.

1. const성원함수

const성원함수는 그 클라스의 성원자료를 절대로 변경하지 않는다는것을 담보한다. 실례 6-12는 이것을 보여준다.

(실례 6-12) const성원함수

```
#include <iostream>
using namespace std;
class AClass
{
private:
   int alpha;
public:
   void NonConFunc() // 비const성원함수
      { alpha = 99; }   // 옳다
   void ConFunc()      // const성원함수
      { alpha = 99; }   // 오유: 성원을 변경할수 없다
};
```

비const함수 NonConc()에서는 성원자료 alpha를 변경할수 있으나 const함수 ConFunc()에서는 변경할수 없다. 그것을 시도하면 번역프로그람은 오유를 통보한다.

함수선언뒤와 함수본체앞에 예약어 const를 배치하여 함수를 const함수로 만들수 있다. 함수선언이 따로 있다면 선언과 정의에 모두 const를 배치해야 한다.

객체로부터 자료를 얻는 성원함수는 자료를 변경할 필요가 없으므로 const로 만들어야 한다.

함수를 const로 만들어서 객체의 자료를 번경하려고 하면 번역프로그람이 오유를 통보하게 하여 그 객체의 어떤 자료도 변경하지 않고 함수를 사용한다는것을 담보할수 있다. 또한 const객체의 창조와 사용도 가능하다.

- Distance실레

한번에 너무 많은 문제들을 제기하지 않기 위하여 지금까지의 실례들에서 const성원함수를 사용하지 않았다. 그러나 const성원함수를 사용할수 있는 경우는 많다. 실례

로 여러개의 실례들에서 소개한 Distance클라스에서 ShowDist()성원함수를 호출할 때에 객체의 어떤 자료도 변경하지 말아야 하므로 const로 만들수 있다. 이 함수는 단순히 자료를 표시한다.

다만 실례 6-13에서 AddDist()함수는 그것을 호출한 객체의 어떤 자료도 변경하지 말아야 하며 그 객체와 인수로서 넘어온 객체를 더하고 값을 돌려주어야 한다. 두개의 const함수의 실행을 고찰하기 위하여 실례 6-9의 프로그람을 변경하였다.

(실례 6-13) const성원함수와 성원함수에로의 const인수

```
#include <iostream>
using namespace std;
class Distance
{
private:
   int meters; float centies;
public:
   Distance() : meters(0), centies(0) { }
   Distance(int me, float ce) : meters(me), centies(ce) { }
   void GetDist()
   {
      cout << "\n메터를 입력하십시오:"; cin >> meters;
      cout << "센치메터를 입력하십시오:"; cin >> centies;
   }
   void ShowDist() const { cout << meters << "m " << centies << "cm"; }
   Distance AddDist(const Distance&) const;
};
Distance Distance::AddDist(const Distance& d2) const
{
   Distance temp;
// meters = 0;        // 오유: 이 자체를 변경할수 없다
// d2.meters = 0;// 오유: d2을 변경할수 없다
   temp.centies = centies + d2.centies; temp.meters = 0;
   if(temp.centies >= 100.0) { temp.centies -= 100.0; temp.meters++; }
   temp.meters += meters + d2.meters;
   return temp;
}
int main()
{
   Distance dist1, dist3; Distance dist2(11, 6.25);
   dist1.GetDist(); dist3 = dist1.AddDist(dist2);
   cout << "\ndist1="; dist1.ShowDist();
   cout << "\ndist2="; dist2.ShowDist();
   cout << "\ndist3="; dist3.ShowDist(); cout << endl;
   return 0;
}
```

여기서 ShowDist()와 AddDist()는 둘다 const성원함수이다. AddDist()의 첫 설명문 meters=0에서 이 const함수를 호출한 객체의 자료를 변경하려고 시도하면 코드자

체가 오유를 통보한다는것을 알수 있다.

- const성원함수의 인수

5장에서 참고에 의하여 어떤 인수를 일반함수에 넘길 때 함수에서 그것을 변경하지 못하게 하려면 함수선언(함수선언과 정의)에서 그 인수를 const로 만들어야 한다는 것을 언급하였다. 이것은 성원함수의 경우에도 같다.

실례 6-13에서 인수는 참고에 의해 AddDist()에로 넘어가고 main()에서 dist2변수를 변경하지 않는다는것을 담보하기 위하여 AddDist()에 대한 인수 d2을 선언과 정의에서 모두 const로 만든다.

두번째 설명문은 번역프로그람이 AddDist()에서 d2의 성원자료를 변경하려는 어떤 시도에나 오유를 통보한다는것을 보여준다.

2. const객체

여러개의 실례프로그람에서 int와 같은 기본형의 변수들을 변경으로부터 방지하기 위하여 const를 적용하였다. 같은 방법으로 클라스의 객체들에 const를 적용할수 있다. 객체를 const로 선언하면 그것을 변경할수 없다. 이 방법이 객체를 변경하지 않는다는 것을 담보하는 유일한 수법이므로 여기서는 const성원함수만 사용할수 있다.

실례 6-14에서 그것을 보여준다.

(실례 6-14) const Distance객체

```
#include <iostream>
using namespace std;
class Distance
{
private:
   int meters;
   float centies;
public:
   Distance() : meters(0), centies(0)   { }
   Distance(int me, float ce) : meters(me), centies(ce) { }
   void GetDist()
   {
      cout << "\n메터를 입력하십시오:";
      cin >> meters;
      cout << "센치메터를 입력하십시오:";
      cin >> centies;
   }
   void ShowDist() const { cout << meters << "m " << centies << "cm"; }
};

int main()
{
   const Distance football(120, 0);
```

```
    // football.GetDist(); // 오유: const성원함수가 아니다
    cout << "\n축구경기장의 길이=";
    football.ShowDist();
    cout << endl;
    return 0;
  }
```

축구경기장은 정확히 120m길이를 가진다. 프로그람에서 축구경기장의 길이를 사용하려면 그것을 const로 만들어야 한다. 실례 6-14에서는 football을 const변수로 만든다. 그러면 ShowDsit()와 같은 유일한 const함수만 이 객체를 호출할수 있다. GetDist()와 같은 비const함수는 사용자로부터 얻은 새로운 값을 객체에 주므로 번역할수 없다. GetDist()함수에서는 football의 const값을 강제로 변경하려고 한다.

클라스를 설계할 때 그 객체의 어떤 자료도 변경하지 말아야 하는 함수를 const로 하는것은 좋은 생각이다. 이것은 클라스사용자가 const객체를 창조하게 한다. const객체는 임의의 const함수를 사용할수 있으나 비const함수를 사용할수 없다. const의 사용은 번역프로그람이 프로그람작성자를 방조하게 한다.

그러면 이것은 무엇을 의미하는가?

지금까지 클라스와 객체에 대하여 소개하였는데 그것들이 실제로 제공하는 리득이 무엇인가를 의심할수 있다. 이 장의 일부 프로그람들을 4장과 비교해보면 수속적수법에 의한 사물현상의 분류를 객체에 의한 분류와 같은 방법으로 할수 있다.

객체지향프로그람작성법의 하나의 우점은 프로그람에 의해 모의되고있는 현실세계의 사물현상과 프로그람의 C++객체들사이의 일정한 일치성이다.

프로그람안의 Part객체는 현실세계의 부분품을 표시하고 카드객체는 카드를, 직4각형 객체는 직4각형 도형을 표시한다. C++에서 부분품에 대한 모든것 즉 부분품번호와 기타 자료항목 그리고 자료호출과 조작에 필요한 함수들을 클라스서술에 포함한다. 이것은 프로그람작성문제를 개념화하기 쉽게 한다. 프로그람의 어느 부분을 객체로서 가장 효과적으로 표시할수 있는가를 알아낸 다음 그 객체와 관련한 자료와 함수를 모두 클라스에 넣는다. C++클라스에 의하여 카드를 표시한다면 카드값을 표시하는 자료항목들을 클라스에 넣고 또한 값을 설정하고 값을 얻고 값을 표시하고 값을 비교하는 함수들도 모두 클라스에 넣는다.

이와 대조적으로 수속적프로그람에서는 현실세계의 객체와 련결된 외부변수와 함수들이 프로그람의 여기저기에 널려있다. 이것들은 쉽게 파악할수 없다.

일부 경우에 현실생활환경의 어떤 부분을 객체로 구성하여야 하는지 명백하지 않다.

실례로 장기프로그람을 쓴다면 무엇이 객체인가? 장기선수, 장기판의 칸, 가능한 장기판의 모든 위치가 객체로 될수 있다.

작은 프로그람은 자주 시행착오(trial and error)적으로 작성할수 있다. 문제를 객

체들로 분할하고 그 객체를 위한 시행클라스의 선언을 쓴다. 만일 클라스들이 현실과 일치하는것 같으면 계속하고 그렇지 않으면 다른 실체들을 클라스로 선택하고 다시 시작해야 한다. 객체지향프로그람작성에 대한 경험이 많을수록 프로그람문제를 클라스들로 분리하기 쉽다.

큰 프로그람을 시행착오방법으로 작성하는것은 너무 복잡하다. 새로운 분야인 객체지향설계가 프로그람문제를 분석하는데 많이 응용되고있으며 현실세계의 사물현상을 표시하기 위하여 어떤 클라스와 객체들을 사용하겠는가를 구상해낸다. 이것을 일반적으로 문제령역(problem domain)이라고 한다.

객체지향프로그람작성의 리득의 일부는 이 절에서 주지 않는다. 객체지향프로그람작성법은 큰 프로그람들의 복잡성에 대응하기 위하여 제기되였다.

작은 프로그람들(즉 이상의 실례들)에서는 객체지향프로그람작성법이 제공해주는 조직적능력이 그리 필요없다. 프로그람이 클수록 그 리득은 더욱더 커진다. 작은 프로그람인 경우에도 객체지향수법으로 고찰을 시작하면 객체지향설계수법은 자연스럽고 놀랄만큼 도움으로 된다. 하나의 우점은 객체지향프로그람작성에서는 수속적프로그람에서보다 번역프로그람이 훨씬 더 많은 개념적오유를 찾을수 있다는것이다.

요 약

클라스는 객체들의 명세서 혹은 설계도이다. 객체는 자료와 그 자료에 대하여 조작하는 함수들로 이루어진다. 클라스선언에서 성원(즉 자료 혹은 함수)들은 비공개 혹은 공개일수 있다. 비공개는 그 클라스의 성원함수들에 의해서만 호출될수 있다는것을 의미하고 혹은 공개는 프로그람안의 임의의 함수에 의해 호출할수 있다는것을 의미한다.

성원함수는 클라스의 성원으로 되는 함수이다. 성원함수들은 객체의 비공개자료를 호출할수 있고 비성원함수들은 그렇게 할수 없다.

구성자는 특수성원함수로서 클라스와 같은 이름을 가지고 그 클라스의 객체가 창조될 때에 실행된다. 구성자는 돌림값이 없고 인수를 가질수 있으며 객체의 자료성원들에 초기값을 주는데 쓰인다. 구성자를 재정의할수 있으므로 객체는 여러가지 방법으로 초기화할수 있다.

해체자는 클라스와 이름이 같은 성원함수이고 앞에 물결표(~)가 붙는다. 해체자는 객체가 해체될 때 호출된다. 해체자는 인수를 가지지 않고 돌림값이 없다.

콤퓨터의 기억기에는 클라스로부터 창조되는 매개 객체에 대한 자료성원들의 개별적인 사본이 있지만 클라스의 성원함수들의 사본은 오직 하나만 있다. 자료항목을 정적으로 만들면 클라스의 모든 객체에 대하여 하나의 실례로 제한할수 있다.

객체지향프로그람작성법을 사용하는 하나의 리유는 현실세계의 객체들과 객체지향

프로그람의 클라스들사이의 밀접한 일치성이다. 프로그람에서 사용하려는 객체와 클라스들의 결정은 복잡해질수 있다. 작은 프로그람에서는 시행착오이면 충분하지만 일반적으로 큰 프로그람에서는 더 계층적인 수법이 필요하다.

문 제

1. 클라스선언의 목적은 무엇인가?

2. 기본자료형이 그 형의 변수사이의 련관과 마찬가지로 객체지향에서는 무엇을 말할수 있는가?

3. 클라스선언에서 비공개로 지정된 자료나 함수는

① 프로그람안의 임의의 함수로부터 호출할수 있다.

② 암호만 알면 호출할수 있다.

③ 그 클라스의 성원함수들에서 호출할수 있다.

④ 클라스의 공개성원들에서만 호출할수 있다.

어느것이 옳은가?

4. int형비공개자료성원 crowbar와 void Pry()라는 선언을 가지는 공개함수를 하나 가지는 Leverage라는 클라스를 창조하는 선언을 쓰시오.

5. 클라스안의 자료항목은 비공개로 하여야 하는가?

6. 문제 4에서 서술한 Leverage클라스의 객체 lever1을 정의하는 명령문을 쓰시오.

7. 점연산자(클라스성원호출연산자)는 다음과 같이 두개의 실체를 련결한다.

① 클라스성원과 클라스객체

② 클라스객체와 클라스

③ 클라스와 그 클라스의 성원

④ 클라스객체와 그 클라스의 성원

어느것이 옳은가?

8. 문제 4, 6에서 서술한 lever1객체의 Pry()함수를 실행하는 명령문을 쓰시오.

9. 클라스선언에서 정의된 성원함수들은 기정으로 inline이다. 옳은가?

10. 문제 4에서 Leverage클라스용의 GetCrow()성원함수를 쓰시오. 이 함수는 crowbar자료값을 돌려주어야 한다. 함수를 클라스선언안에서 정의한다고 가정하시오.

11. 구성자는 어느때 실행되는가?

12. 구성자의 이름은 무엇의 이름과 같은가?

13. 문제 4에서의 Leverage클라스의 성원인 crowbar자료를 0으로 초기화하는 구성자를 쓰시오. 구성자는 클라스선언에서 정의하시오.

14. 클라스는 같은 이름의 구성자를 한개이상 가질수 있는가?

15. 성원함수는 항상 다음 자료를 호출할수 있다.

① 함수가 성원으로 되는 객체의 자료

② 함수가 성원으로 되는 클라스의 자료

③ 함수가 성원으로 되는 클라스의 임의의 객체의 자료

④ 그 클라스의 공개부분의 자료

어느것이 옳은가?

16. 문제 10에서 서술한 성원함수 GetCrow()를 클라스선언밖에서 정의한다고 가정하고 선언을 클라스선언안으로 가져가시오.

17. 문제 10으로부터 클라스선언밖에서 정의하는 GetCrow()성원함수의 갱신판을 쓰시오.

18. C++의 구조체와 클라스의 중요한 기술적차이는 무엇인가?

19. 클라스의 세개의 객체를 정의한다면 기억기에는 그 클라스의 자료항목의 사본이 몇개, 성원함수의 사본이 몇개 보관되는가?

20. 객체에 통보문을 보내는것이 C++에서는 무엇과 같은가?

21. 클라스가 유익한것은

① 그것을 사용하지 않을 때 기억기에서 제거되기때문이다.

② 자료를 다른 클라스로부터 은페하기때문이다.

③ 한개 실체의 모든 특성을 하나로 융합하기대문이다.

④ 현실세계의 객체들을 거의 근사적으로 모형화할수 있기때문이다.

어느것이 옳은가?

22. 현실세계의 프로그람작성문제를 클라스로 분할하기 위한 간단하면서 정확한 방법론이 있다. 옳은가?

23. const성원함수는 그것을 호출하는 객체에 대하여

① const와 비const성원자료를 변경할수 있다.

② 오직 const성원자료만 변경할수 있다.

③ 오직 비const성원자료만 변경할수 있다.

④ const와 비const성원자료를 변경할수 없다.

어느것이 옳은가?

24. const객체를 선언하는 const성원함수들과만 사용할수 있다. 옳은가?

25. float형의 jerry라는 const인수를 하나 가지는 AFunc()라는 const void함수의 선언을 쓰시오.

련습문제

1. 기본자료형int의 기본기능을 모의하는 클라스 Int를 창조하시오. Int클라스의 유

일한 자료는 int변수이다. Int를 0으로 초기화하는 성원함수, 그것을 int값으로 초기화하는 성원함수, 둘째 Int값을 더하는 성원함수를 정의하시오. 두개의 초기화된 Int변수와 하나의 초기화하지 않은 Int변수를 창조하고 두개의 초기화된 값들을 더하고 결과를 초기화되지 않은 변수에 대입하고 결과를 표시하시오.

2. 다리의 통과료를 추상화하시오. 다리를 지나는 자동차들은 통과료로서 50전을 받는다. 때때로 자동차들은 통과료를 지불하지 않고 지나간다. 클라스 Toolbooth는 지나가는 차량들의 수(unsigned int)와 받은 통과료의 총액(double)을 보관한다. 구성자는 이것들을 령으로 초기화한다. PayingCar()성원함수는 전체 차량수를 증가시키고 총통과료에 0.50을 더한다. NoPayingCar()성원함수는 차량수를 증가시키지만 총 통과료를 증가시키지 않는다. 끝으로 Display()성원함수는 두개의 량들을 표시한다. 적당한 성원함수를 const로 하시오. Toolbooth클라스를 시험하는 프로그람을 쓰시오. 이 프로그람은 통과료를 지불하는 차량을 계수하기 위하여 어떤 건을 사용자가 누르게 하고 또한 통과료를 지불하지 않는 차량을 계수하기 위해 다른 건을 누르게 하여야 한다. Esc건을 누르면 프로그람은 총 차량수와 총액을 출력하고 완료한다.

3. 시, 분, 초로서 각각 int형성원자료를 가지는 Time이라는 클라스를 창조하시오. 한개 구성자는 자료들을 0으로 초기화하고 다른 구성자는 주어진 값으로 초기화한다. 다른 성원함수는 시간을 12:59:59형식으로 표시한다. 마지막 성원함수는 인수로서 넘어온 Time형의 객체를 두개 더한다. main()함수는 두개의 초기화된 Time객체(const이여야 한다)들을 창조하고 초기화하지 않은것을 하나 창조한다. 그다음 두개의 초기화된 값들을 더하여 결과를 셋째 Time변수에 보관하고 그 값을 표시한다. 적당한 성원함수를 const로 하시오.

4. 4장 련습 4에 기초하여 Employee클라스를 창조하시오. 성원자료는 종업원번호를 보관하는데 int, 종업원의 생활비를 보관하는데 float를 사용한다. 성원함수는 사용자로부터 이 자료들을 읽어들인다. 사용자가 세명의 종업원의 자료를 입력하고 표시하는 main()을 정의하시오.

5. 4장 련습 5의 Date구조체에 기초하여 Date클라스를 창조하시오. 자료성원으로서 세개의 int형 년, 월, 일을 가진다. 또한 두개의 성원함수 즉 날자를 97/12/31형식으로 입력하는 GetDate()와 날자를 표시하는 ShowDate()가 있다.

6. Date클라스와 EType렬거형(4장 련습 6)을 포함하도록 련습 4의 Employee클라스를 확장하시오. Date클라스의 객체는 우선 입직한 날자를 보관해야 한다. EType변수는 종업원의 형 즉 Laborer, Manager, Researcher 등을 보관한다. 두개의 항목들은 Employee선언에서 비공개성원이다. 사용자로부터 새 정보를 얻고 표시하는데 사용하는 GetEmployee()와 PutEmployee()함수를 확장해야 한다. 이 함수들은 switch명령문을 사용하여 EType변수를 조종한다. 사용자가 세개의 Employee변수용 자료를 입력하게 하고 그것을 표시하는 main()함수를 쓰시오.

7. 대양을 항행할 때 배의 위치는 위도와 경도의 도와 분으로 표시한다. 실례로 동경 149도 34.8분, 북위 17도 31.5분을 149° 34.8'W, 17° 31.5'라고 쓴다. 1° 는 60'이다.(낡은 방법은 분을 다시 60"로 나누지만 현대수법에서는 그대신 10진수 분을 사용한다.) 경도(longitude)는 0～180° 로 표시하고 영국의 그리니치로부터 동쪽인가, 서쪽인가에 따라서 동경과 서경으로 표시한다. 또한 위도(latitude)는 0～90° 로 표시하고 적도로부터 북극 또는 남극으로 가면서 북위 또는 남위로 표시한다. 세개의 성원변수 int형degree, float형minute, 방향문자용 char(N,S,E,W)를 가지는 클라스 Angle을 창조하시오. Angle클라스는 위도변수 혹은 경도변수를 보관할수 있다. Angle값(° 와 ')을 얻는 성원함수, 그것을 179° 59.9'E형식으로 출력하는 성원함수를 쓰시오. 또한 3인수 구성자를 쓰시오. 구성자에 의해 초기화된 각도를 가지는 객체를 창조하고 순환안에서 사용자가 각도값을 입력하게 하고 값을 표시하는 main()프로그람을 쓰시오. 보통 도(°)기호를 출력하는 16진문자상수 '\xF8'을 사용할수 있다.

8. 클라스로부터 창조한 매 객체의 계렬번호를 보관하는 자료성원을 포함하는 클라스를 창조하시오. 즉 창조된 첫째 객체는 1, 둘째는 2, …으로 번호를 붙인다. 그러자면 지금까지 그 클라스로부터 몇개의 객체가 창조되였는가를 기록하는 자료성원이 있어야 한다.

이 성원을 개별적인 객체가 아니라 모든 클라스에 적용해야 한다면 어떤 예약어를 지정해야 하는가?

그다음 매개 객체를 창조하는데 구성자는 이 const성원변수를 사용하여 새 객체용의 적당한 계렬번호를 결정한다. 객체가 자기의 계렬번호를 표시하는 성원함수를 추가하시오. 그다음 세개의 객체를 창조하고 그 매개의 계렬번호를 표시하는 main()프로그람을 쓰시오. 그것들은 "나는 2번 객체입니다"라고 표시한다.

9. 4장의 련습 8의 Fraction구조체를 클라스로 변환하시오. 성원자료는 분수의 분자와 분모이다. 성원함수는 3/5형식으로 사용자로부터 입력을 받아들이고 같은 형식으로 분수의 값을 출력한다. 다른 성원함수는 두개의 분수값을 더한다. main()프로그람은 두개의 분수를 반복하여 입력하고 그 합을 표시한다. 매번 조작후에 사용자에게 계속하겠는가를 묻는다.

10. 배의 번호와 위치를 성원으로 가지는 Ship클라스를 창조하시오. Ship객체의 수를 계수하는데 련습 8의 방법을 사용하시오. 배의 위도와 경도위치에 련습 7의 Angle 클라스의 변수를 두개 사용하시오. Ship클라스의 하나의 성원함수는 사용자로부터 위치를 얻어서 그 객체에 보관하고 다른 성원함수는 계렬번호와 위치를 표시한다. Ship 객체들을 창조하고 사용자가 매개 위치를 입력하게 하고 매 배의 번호와 위치를 표시하는 main()프로그람을 쓰시오.

11. 5장 련습 12의 4기능분수수산기를 구조체가 아니라 Fraction클라스로 변경하시오. 4가지 산수연산은 물론 입력과 출력을 위한 성원함수들이 있어야 한다. 또한 분

수를 최소항까지 약분하는 기능이 있어야 한다. 여기에 성원인 Fraction객체를 최소항으로 약분하는 성원함수가 있다. 그것은 분수의 분자와 분모의 최대공약수(gcd)를 구하여 두 수를 나누는데 사용한다.

```
void Fraction::LowTerms()
{
    long tnum, tden, temp, gcd;
    tnum = labs(num);   // 부 아닌 부분을 사용
    tden = labs(den);
    if(tden == 0)              // n/0인가 검사
    {
        cout << “분수오유: 0에 의한 나누기\n”;
        exit(1);               // 0/n인가 검사
    }
    else if(tnum == 0)
    {
        num = o;
        den = 1;
        return;
    }
    // 순환은 tnum과 tden의 최대공약수를 찾는다
    while(tnub != 0)
    {
        if(tnum < tden) // 분모가 더 큰가를 검사
        {
            temp = tnum;
            tnum = tden;
            tden = temp;
        }
        tnum = tnum - tden;
    }
    gcd = tden;                // 최대공약수
    num = num / gcd;           // 약분한다
    den = den / gcd;
}
```

이 함수는 매 산수함수의 끝에서 호출하거나 출력하기 전에 호출할수 있다. 또한 4개의 산수연산과 입력과 표시를 위한 성원함수와 2인수구성자가 있어야 한다.

12. 객체지향프로그람작성법의 우점은 클라스전체를 변경하지 않고 서로 다른 프로그람에서 사용할수 있는것이다. 분수의 곱하기표를 생성하는 프로그람에서 련습 11로부터 Fraction클라스를 사용하시오. 사용자가 나누이는 수를 입력하게 하고 0과 1사이에 있는 두개 분수의 결합을 모두 생성하여 그것들을 모두 곱하시오. 아래에 나누는 수가 6인 경우의 출력이 있다.

```
      1/6    1/3    1/2    2/3    5/6
1/6   1/36   1/18   1/12   1/9    5/36
1/3   1/18   1/9    1/6    2/9    5/18
```

1/2	1/12	1/6	1/4	1/3	5/12
2/3	1/9	2/9	1/3	4/9	5/9
5/6	5/36	5/18	5/12	5/9	25/36

제 7 장. 배렬과 문자렬

프로그람언어에서는 같은 형의 자료항목들을 모두 묶을 필요가 있다. C++에서 이것을 실현하는 가장 기초적인 기구는 배렬이다. 배렬(array)은 여러개 또는 수만개의 자료항목을 보관할수 있다. 배렬에 묶어진 자료항목들은 int, float와 같은 기본형이거나 구조체와 객체와 같은 사용자정의형일수 있다.

배렬은 많은 항목들을 더 큰 단위로 묶는데서 구조체와 같다. 그러나 구조체가 일반적으로 다른 형의 항목들을 묶는다면 배렬은 같은 형의 항목들을 묶는다.

중요한것은 구조체의 항목들은 이름에 의해 호출되지만 배렬의 항목들은 첨수에 의하여 호출되는것이다. 첨수에 의한 항목의 지정은 많은 량의 항목들을 쉽게 호출할수 있게 한다. 배렬은 거의 모든 프로그람언어에 있다. C++의 배렬은 다른 언어의 배렬과 비슷하며 C의 배렬과 같다.

이 장에서는 우선 int, char와 같은 기본자료형의 배렬을 고찰한다. 그다음 클라스의 자료성원으로 사용되는 배렬을 시험하고 객체들을 보관하는 배렬들을 시험한다. 이리하여 이 장에서는 배렬을 소개할뿐아니라 객체지향프로그람작성법의 리해를 도모한다.

표준 C++에서 배렬은 같은 형의 원소들을 묶는 유일한 방법이 아니다. 벡토르는 표준형판서고의 부분으로서 같은 형의 원소들을 묶는 다른 수법이다.

이 장에서는 또한 본문을 보관하고 조작하는데 쓰이는 문자렬에 대한 두가지 조작방법을 고찰한다. 첫째 종류의 문자렬은 char형의 배렬이고 둘째 종류는 표준 C++ string클라스의 객체이다.

제 1 절. 배렬의 기초

다음의 간단한 실례는 배렬을 소개한다. 실례 7-1은 4명의 나이를 보관하는 옹근수배렬을 창조한다. 그리고 사용자에게 4개 값을 입력하게 하고 그것을 배렬에 보관한다. 끝으로 4개 값을 모두 표시한다.

(실례 7-1) 4명의 나이를 입력하고 화면에 표시한다.

```
#include <iostream>
using namespace std;
int main()
{
   int age[4];
   for(int j=0; j<4; j++)
   {
      cout << "나이를 입력하시오: ";
```

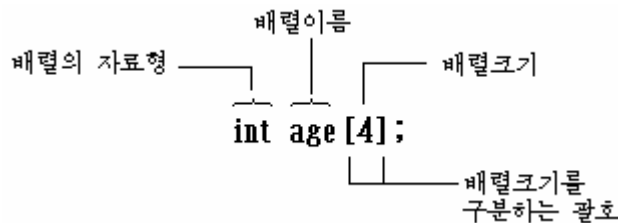
```
        cin >> age[j];
    }
    for(j=0; j<4; j++)
        cout << "나이는 " << age[j] << endl;
    return 0;
}
```

처음의 for순환은 사용자로부터 나이를 얻어서 배렬에 보관하고 두번째 순환은 그것들을 배렬로부터 읽어들여 표시한다.

1. 배렬의 정의

C++의 다른 변수처럼 배렬은 정의한 다음 정보를 보관하는데 사용할수 있다. 그리고 다른 정의들에서처럼 배렬정의는 변수형과 이름을 지정한다. 또한 배렬의 또 하나의 특성으로서 크기를 포함한다. 크기는 배렬에 보관할수 있는 자료항목의 량을 지정한다. 크기는 이름뒤의 중괄호 []안에 들어있다. 그림 7-1은 배렬정의의 문법을 보여준다.

그림 7-1. 배렬정의문법

2. 배렬원소

배렬안의 항목들을 원소(element)라고 부른다. 이것은 성원이라고 부르는 구조체의 항목과 대조된다. 배렬의 모든 원소들은 같은 형으로 되여있고 값들만 변한다. 그림 7-2는 배렬 age의 원소들을 보여준다.(그림에서 int형은 16bit체계에서와 같이 2byte를 처리하는것으로 가정한다.)

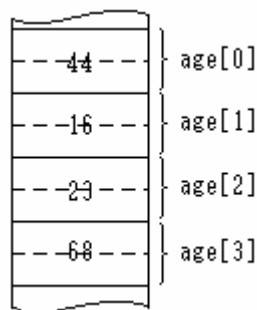



그림 7-2. 배렬원소

관례에 따르면 기억기는 그림에서 아래쪽으로 커진다. 즉 첫 배렬원소는 기억기의 웃끝에 놓이고 그다음의 원소들은 아래쪽으로 전개된다.

age의 매개 원소가 옹근수이므로 2byte(16bit체계)를 차지한다. 정의에서 지적한바와 같이 배렬은 정확히 4개 원소를 가진다.

배렬의 첫째 원소는 번호 0을 가진다. 원소가 4개 있으므로 마지막 원소의 번호는 3이다.

3. 배렬원소의 호출

실례 7-1에서는 매개 배렬원소를 두번씩 호출한다. 처음으로 다음행

```
cin >> age[j];
```

에서 호출하고 배렬에 값을 삽입한다.

두번째 호출은 다음행

```
cout << “나이는 ” << age[j];
```

에 있고 배렬에서 값을 읽어낸다.

배렬원소의 호출은 배렬의 이름과 변수 j를 구분하는 괄호들로 구성된다. 이 식에 의해 4개 배렬원소들중 어느것이 지정되는가 하는것은 j의 값에 의존한다. 즉 age[0]은 첫 원소, age[1]은 둘째 원소, age[2]는 세번째 원소, age[3]은 네번째원소를 참고한다. 괄호안의 변수(또는 상수)를 배렬첨수(index)라고 부른다.

j는 두개의 for순환에 있는 순환변수로서 0으로 시작하여 3에 이를 때까지 증가되면서 배렬원소들을 차례로 호출한다.

- 배렬원소의 평균값계산

실례 7-2는 사용자가 1주간의 매일(일요일 제외)의 판매액을 표시하는 6개 값들의 배렬을 입력하게 하고 이 값들의 평균을 계산한다. 판매액을 보관하는데 double형배렬을 사용한다.

(실례 7-2) 한주일동안 상품의 평균판매액계산

```
#include <iostream>
using namespace std;
int main()
{
   const int SIZE = 6;
   double sales[SIZE];
   cout << "6일간의 매일 판매액을 입력하시오\n";
   for(int j=0; j<SIZE; j++)
      cin >> sales[j];
   double total = 0;
   for(j=0; j<SIZE; j++)
      total += sales[j];
   double average = total / SIZE;
   cout << "평균 판매액=" << average << endl;
   return 0;
}
```

이 프로그람에서 새로운것은 배렬크기에 const변수의 사용과 순환제한의 사용이다.

const변수는 프로그람의 선두에서 정의된다.

```
const int SIZE = 6;
```

앞의 실례와 같이 4와 같은 수값대신에 변수를 사용하면 배렬크기를 쉽게 변경할 수 있다. 오직 한개의 프로그람행을 변경할뿐 배렬크기, 순환제한, 그밖의 배렬크기를 사용하는 모든 곳에서 크기를 변경할수 있다. 모두 대문자로 되여있는 이름을 가진 변수는 프로그람에서 변경할수 없는 상수로 생각하면 된다.

4. 배렬의 초기화

배렬을 처음 정의할 때 매개 배렬원소들에 값을 줄수 있다. 다음의 실례 7-3은 배렬 daysPerMouth의 12개 배렬원소에 매개 달의 날자수를 설정한다.

(실례 7-3) 정월초하루부터 주어진 날까지의 날자수계산

```
#include <iostream>
using namespace std;
int main()
{
   int month, day, totalDays;
   int daysPerMonth[12] = { 31,28,31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
   cout << "\n월을 입력하시오(1-12):";
   cin >> month;
   cout << "일을 입력하시오(1-31):";
   cin >> day;
   totalDays = day;
   for(int j=0; j<month-1; j++)
      totalDays += daysPerMonth[j];
   cout << "정월초하루부터 총 날자수=" << totalDays << endl;
   return 0;
}
```

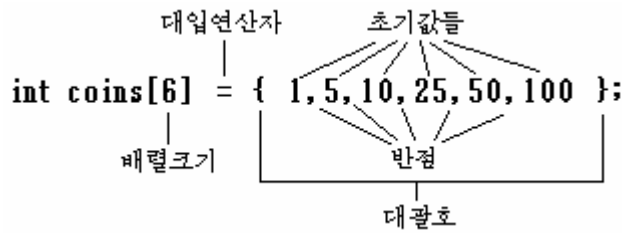
프로그람은 먼저 첫날부터 사용자가 지정한 날까지의 날자수를 계산한다.(윤년은 고려하지 않는다.) 여기에 프로그람과의 대화가 있다.

```
월을 입력하시오(1-12):3
일을 입력하시오(1-31):11
정월초하루부터 총 날자수=70
```

월과 일의 값을 얻으면 프로그람은 우선 totalDays변수에 일값을 대입하고 순환하면서 daysPerMonth배렬로부터 totalDays까지 값들을 더한다.

가능한 값들의 수는 월의 수보다 하나 적다. 실례로 사용자가 5월을 입력한다면 처음 4개 배렬원소(31, 28, 31, 30)의 값들이 총계를 보관하는 변수에 더해진다.

daysPerMonth의 초기값들은 괄호안에 들어있고 반점에 의해 구분된다. 그것들은 같기기호에 의해 배렬식과 련결된다. 그림 7-3은 그 문법을 보여준다.

그림 7-3. 배렬초기화문법

배렬원소들을 모두 초기화할 때에는 배렬크기가 요구되지 않는다. 이때 번역프로그람은 초기값들을 계수하여 배렬크기를 결정한다. 따라서 다음과 같이 쓸수 있다.

```
int daysPerMonth[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

그러면 배렬크기를 명백히 지정하였는데 초기값들의 수와 크기가 일치하지 않으면 어떤 일이 생기는가?

초기화자의 수가 배렬크기보다 적으면 빠진 원소들은 0으로 설정된다. 그러나 초기값이 배렬크기보다 많으면 오유가 경고된다.

5. 다차원배렬

지금까지는 1차원배렬을 고찰하였다. 여기서는 한개의 변수가 매개의 배렬원소를 지정한다. 그러나 배렬은 더 큰 차원을 가질수 있다. 실례 7-4는 2차원배렬을 사용하여 여러 지역에서의 몇달동안의 판매액자료를 보관한다.

(실례 7-4) 2차원배렬을 사용하여 상품의 평균판매액표시

```
#include <iostream>
#include <iomanip>
using namespace std;
const int DISTRICTS = 4;
const int MONTHS = 3;
int main()
{
   int d, m;
   double sales[DISTRICTS][MONTHS];
   cout << endl;
   for(d=0; d<DISTRICTS; d++)
      for(m=0; m<MONTHS; m++)
      {
          cout << d + 1 << "지역에서 "
               << m + 1 << "월 판매액을 입력하시오: ";
          cin >> sales[d][m];
      }
   cout << "\n\n";
   cout << "                        월\n";
   cout << "             1          2          3\n";
   for(d=0; d<DISTRICTS; d++)
   {
```

```
        cout << "\n지역" << d + 1;
        for(m=0; m<MONTHS; m++)
            cout << setiosflags(ios::fixed) << setiosflags(ios::showpoint)
                 << setprecision(2) << setw(10) << sales[d][m];
    }
    cout << endl;
    return 0;
}
```

이 프로그램은 사용자로부터 판매액자료를 받아들이고 그것을 표로 표시한다.

```
1지역에서 1월 판매액을 입력하시오:3964.23
1지역에서 2월 판매액을 입력하시오:4135.87
1지역에서 3월 판매액을 입력하시오:4397.98
2지역에서 1월 판매액을 입력하시오:867.75
2지역에서 2월 판매액을 입력하시오:923.59
2지역에서 3월 판매액을 입력하시오:1037.01
3지역에서 1월 판매액을 입력하시오:12.77
3지역에서 2월 판매액을 입력하시오:378.32
3지역에서 3월 판매액을 입력하시오:798.32
4지역에서 1월 판매액을 입력하시오:2983.53
4지역에서 2월 판매액을 입력하시오:3983.73
4지역에서 3월 판매액을 입력하시오:9494.98
                    월
             1          2          3
지역1    3964.23    4135.87    4397.98
지역2     867.75     923.59    1037.01
지역3      12.77     378.32     798.32
지역4    2983.53    3983.73    9494.98
```

- 다차원배렬의 정의

닫긴 중괄호안에 두개의 크기지적자가 있는 배렬을 정의하자. 즉

```
double sales[DISTHICTS][MONTHS];
```

sales는 바둑판처럼 배치된 2차원배렬이나 배렬들의 배렬로서 생각할수 있다. 즉 매개가 MONTHS원소들의 배렬인 DISTRICTS원소들의 배렬이다. 그림 7-4는 이것을 고찰하는 방법을 보여준다.

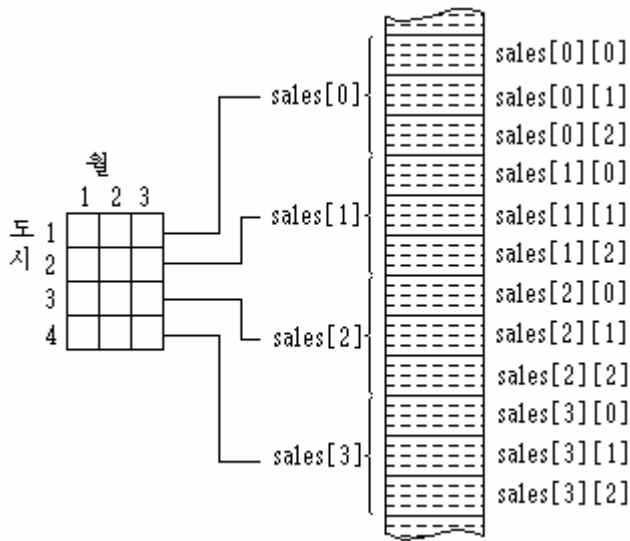



그림 7-4. 2차원배렬

물론 2차원이상의 배렬도 있다.

3차원배렬은 세개의 첨수에 의하여 호출한다.

```
elem = dimen3[x][y][z];
```

이것은 1차원과 2차원배렬들과 거의 같다.

- 다차원배렬원소의 호출

2차원배렬의 배렬원소는 두개의 첨수를 요구한다.

```
sales[d][m];
```

여기서 매개 첨수는 자기의 괄호를 가지고 반점은 쓰이지 않는다. C를 제외한 일부 언어들처럼 sales[d,m]이라고 쓰면 안된다.

- 수를 쓰는 형식의 지정

실례 7-4는 판매액의 표를 표시한다. 여기서는 값들이 적당한 형식으로 표시된다. 판매액값들은 소수점아래 두자리를 가지며 행의 매개 칸에 있는 수값들에 모두 소수점을 붙이고 소수점아래의 모자라는 수자에는 0을 채워서 즉 79.5가 아니라 79.50으로 하려고 한다.

그러자면 C++입출력스트림과 관련한 몇가지 작업이 요구된다. 이미 조작자 setw()를 출력마당폭을 설정하는데 사용하였다. 소수의 출력형식지정은 몇가지 추가적인 조작자를 요구한다.

여기에 10문자폭의 마당에 fpn이라는 류동소수점수를 출력하는 명령문이 있다.

```
cout << setiosflags(ios::fixed)          // 고정소수점수를 출력하는 명령문이 있다.
     << setiosflags(ios::showpoint)      // 항상 소수점을 표시한다.
     << setprecision(2)                  // 소수부는 두자리
     << setw(10)                         // 마당폭 10
     << fpn;                             // 수값
```

ios클라스에는 1bit출력형식지정기발들의 묶음이 long int로 보관되여있는데 이것

은 수행하려는 출력형식지정방법을 결정한다. 현재는 ios클라스가 무엇을 하는지, 이 클라스가 사용하는 문법의 정확한 근거, 조작자가 어떻게 작업하는지 알 필요는 없다.

ios의 두개의 기발 즉 fixed와 showpoint는 서로 련관되여있다. 조작자 setiosflags는 기발의 이름을 인수로 사용하여 그 기발들을 설정한다. 기발의 이름은 클라스이름 ios와 범위해결(scope resolution)연산자 ::뒤에 놓인다.

cout명령문에서 처음의 두 행은 ios기발들을 설정한다. 후에 설정을 해제할 필요가 있으면 resetiosflags조작자를 사용한다. fixed기발은 수를 3.45e3과 같은 지수형식이 아니라 고정소수형식으로 출력하게 한다. showpoint기발은 소수점을 항상 표시한다는 것을 지정한다. 이때 소수부가 없는 수 즉 123은 123.00으로 표시된다.

소수점아래 두자리 정확도로 설정하자면 자리수를 인수로 하는 setprecision조작자를 사용한다. 이미 setw조작자를 사용하여 마당폭을 설정하는 방법을 보았다.

이러한 조작자들이 모두 cout에 전송되여야 수 자체를 요구되는 형식으로 표시할 수 있다.

ios형식기발에 대하여서는 12장에서 취급한다.

- 다차원배렬의 초기화

다차원배렬도 초기화할수 있다. 한가지 미리 알아두어야 할것은 많은 괄호와 반점을 입력해야 하는 점이다. 실례 7-4를 변경한것으로서 사용자에게 입력을 요구하지 않고 초기화된 배렬을 사용하는 실례 7-5를 보기로 하자.

(실례 7-5) 2차원배렬의 초기화

```
#include <iostream>
#include <iomanip>
using namespace std;
const int DISTRICTS = 4;
const int MONTHS = 3;
int main()
{
  int d, m;
  double sales[DISTRICTS][MONTHS] = {
    { 1432.07, 234.50, 654.01 },
    { 322.00, 13838.32, 17589.88 },
    { 9328.34, 934.00, 4172.30 },
    { 12838.29, 2332.63, 32.93 }
  };
  cout << "\n\n";
  cout << "                    월\n";
  cout << "          1         2         3\n";
  for(d=0; d<DISTRICTS; d++)
  {
     cout << "\n지역" << d + 1;
     for(m=0; m<MONTHS; m++)
        cout << setiosflags(ios::fixed)
```

```
                << setiosflags(ios::showpoint)
                << setprecision(2)
               << setw(10)
               << sales[d][m];
    }
    cout << endl;
    return 0;
  }
```

2차원배렬이란 사실상 배렬의 배렬이다. 2차원배렬의 초기화형식은 바로 이러한 사실에 기초하고있다. 매개 부분배렬의 초기값들은 대괄호안에 놓이고 반점에 의하여 구분된다.

```
{ 1432.07, 234.50, 654.01 }
```

기본배렬의 원소인 부분배렬 4개는 모두 대괄호안에 들어있고 반점에 의해 구분된다.

제 2 절. 함수에 배렬의 넘기기

배렬을 함수인수로서 사용할수 있다. 매개 지역의 판매액자료를 표로 표시하는 함수에 배렬을 넘기는 실례 7-5를 변경한 실례 7-6이 있다.

(실례 7-6) 인수로서 배렬의 넘기기

```
#include <iostream>
#include <iomanip>
using namespace std;
const int DISTRICTS = 4;
const int MONTHS = 3;
void Display(double[DISTRICTS][MONTHS]);
int main()
{
  double sales[DISTRICTS][MONTHS] = {
    { 1432.07, 234.50, 654.01 },
    { 322.00, 13838.32, 17589.88 },
    { 9328.34, 934.00, 4172.30 },
    { 12838.29, 2332.63, 32.93 }
  };
  Display(sales);
  cout << endl;
  return 0;
}
void Display(double funSales[DISTRICTS][MONTHS])
{
  int d, m;
  cout << "\n\n";
  cout << "                    월\n";
  cout << "          1         2         3\n";
```

```
        for(d=0; d<DISTRICTS; d++) {
            cout << "\n지역" << d + 1;
            for(m=0; m<MONTHS; m++)
                cout << setiosflags(ios::fixed) << setiosflags(ios::showpoint)
                    << setprecision(2)  << setw(10) << funSales[d][m];
        }
    }
```

1. 배렬인수를 가지는 함수선언

함수선언에서 배렬인수는 배렬의 자료형과 크기에 의해 표시된다. 여기에 Display()함수의 선언이 있다.

```
void Display(float[DISTRICTS][MONTHS]);    //선언
```

실제로 여기에는 불필요한 정보가 있다. 다음 명령문도 우의 명령문처럼 잘 동작한다.

```
void Display(float[][MONTHS]);
```

그러면 함수에서 1차원의 크기가 왜 필요하지 않은가?

2차원배렬은 배렬들의 배렬이다. 함수는 먼저 인수를 지역의 배렬로서 고찰한다. 함수는 지역이 몇개인지 알 필요는 없으나 매개의 지역원소가 얼마나 큰가 하는것을 알아야 한다. 그래야 원소당 byte수에 첨수를 곱하여 지정된 원소가 어디에 있는가를 계산할수 있다. 매개 원소의 크기 MONTHS를 알아야 하지만 그것이 몇개인가 하는 DISTRICTS는 알 필요가 없다. 따라서 인수로서 1차원배렬을 사용하는 함수를 선언한다면 배렬크기를 사용할 필요가 없다.

```
void SomeFunc(int elem[]);    //선언
```

2. 배렬인수를 가지는 함수호출

함수를 호출할 때에는 오직 배렬의 이름만 인수로서 사용한다.

```
Display(sales);        //함수호출
```

사실 배렬이름은 배렬의 기억주소를 표시한다. 주소는 10장에서 고찰한다.

배렬인수에 주소를 사용하는 방법은 참고인수의 경우와 비슷하고 이때 배렬원소의 값들은 함수에 복사되지 않는다. 배렬이 다른 이름으로 참고되여도 함수는 원시배렬과 작업한다.

배렬을 호출하는 매개 함수에서 전체 배렬의 복사는 시간과 기억기를 소비하고 배렬이 매우 커질수 있으므로 이 방법을 사용한다.

그러나 주소는 참고와 다르다. 함수선언에서는 &기호를 배렬이름과 함께 쓰지 않는다. 지적자를 론의할 때까지는 배렬을 그 이름만 가지고 넘기고 함수는 사본이 아니라 원시배렬을 호출하는것으로 리해한다.

3. 배렬인수를 가지는 함수정의

함수정의에서 선언자는 다음과 같다.

```
void Display(double funSales[DISTRICTS][MONTHS])
```

배렬인수는 배렬의 자료형과 이름, 차원의 크기를 사용한다. 함수에 의해 사용된 배렬이름은 배렬을 정의하는 이름과 다를수 있으나 둘다 같은 배렬을 참고한다. 함수가 배렬원소를 적당히 호출하기 위해 차원을 요구하므로 배렬의 모든 차원을 지정하여야 한다.(일부 경우 제1차원은 제외된다.)

함수안에서 배렬원소를 참고할 때에 함수의 배렬인수이름을 사용한다.

```
funSales[d][md]
```

그러나 다른 방법들에서 함수는 마치도 배렬이 함수안에서 정의된것처럼 배렬원소들을 호출할수 있다.

제 3 절. 구조체배렬

배렬은 기본자료형은 물론 구조체도 포함할수 있다. 여기에 4장의 Part구조체에 기초한 실례가 있다.

(실례 7-7) 배렬원소로서의 구조체변수

```
#include <iostream>
using namespace std;
const int SIZE = 4;       // 배렬의 원소수
struct Part
{
  int modelNumber;        // 부분품의 식별번호
  int partNumber;         // 부분품번호
  float cost;             // 부분품의 단가
};
int main()
{
  int n;
  Part aPart[SIZE];       // 구조체배렬정의
  for(n=0; n<SIZE; n++)
  {
     cout << endl;
     cout << "형번호를 입력하시오: "; cin >> aPart[n].modelNumber;
     cout << "부분품번호를 입력하시오: "; cin >> aPart[n].partNumber;
     cout << "단가를 입력하시오: "; cin >> aPart[n].cost;
  }
  cout << endl;
  for(n=0; n<SIZE; n++)
  {
     cout << "형번호=" << aPart[n].modelNumber;
     cout << ", 부분품번호=" << aPart[n].partNumber;
```

```
        cout << ", 단가=" << aPart[n].cost << "원\n";
    }
    return 0;
}
```

사용자는 형번호와 부분품번호, 단가를 입력한다. 프로그램은 구조체에 이 자료들을 보관한다. 그러나 이 구조체는 구조체배렬안의 하나의 원소이다. 프로그램은 4개의 서로 다른 부분품들에 대한 자료를 얻어서 Part배렬의 4개원소들에 보관한 다음 그 정보를 표시한다. 여기에 견본입력이 있다.

```
형번호를 입력하시오: 44
부분품번호를 입력하시오: 4954
단가를 입력하시오: 123.45
형번호를 입력하시오: 44
부분품번호를 입력하시오: 4954
단가를 입력하시오: 123.45
형번호를 입력하시오: 44
부분품번호를 입력하시오: 4954
단가를 입력하시오: 123.45
형번호를 입력하시오: 44
부분품번호를 입력하시오: 4954
단가를 입력하시오: 123.45
```

구조체의 배렬은 명령문

```
Part aPart[SIZE];
```

에서 정의된다.

이것은 기본자료형의 배렬과 같은 문법을 가진다. 다만 형이름 Part는 이것이 더 복잡한 형의 배렬이라는것을 보여준다.

배렬의 원소로 되는 구조체의 성원인 자료항목의 호출에서는 새로운 문법을 사용한다. 례를 들면

```
aPart[n].modelNumber;
```

는 aPart배렬의 첨수가 n인 구조체의 modelNumber성원을 참고한다. 그림 7-5는 기억기에서 구조체배렬의 보관방법을 보여준다.

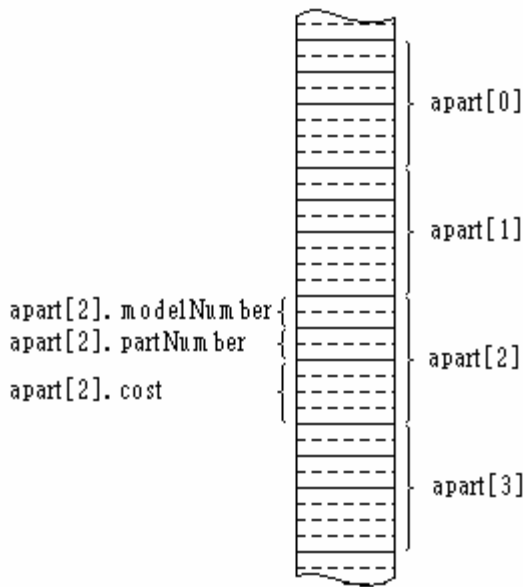



그림 7-5. 구조체배렬

구조체배렬은 여러가지 목적에 사용할수 있는 자료형이므로 종업원자료(이름, 나이 등)의 배렬, 도시의 지형학적자료(이름, 인구 등)의 배렬, 기타 다른 자료형들의 배렬도 보관할수 있다.

제 4 절. 클라스성원자료로서 배렬

배렬을 클라스의 자료항목으로 사용할수 있다. 탄창을 일반적인 콤퓨터자료구조로서 모형화하는 실례를 고찰하자.

탄창은 총에 탄알들을 보관하는 용수적재장치처럼 작업한다. 꼭대기에 탄알을 밀어넣을 때 탄창은 아래로 내려가고 탄알을 꺼낼 때에는 우로 올라온다. 탄창에 마지막으로 넣은 탄알은 항상 처음에 나온다.

이미 언급한것처럼 함수는 자기 인수들을 넘길 때 그 귀환주소를 탄창에 보관한다. 이와 같은 탄창은 하드웨어에 의해 부분적으로 실현되고 기호언어에서 가장 많이 호출된다. 또한 탄창은 쏘프트웨어에 의해서도 창조될수 있다. 쏘프트웨어탄창은 대수식을 해석하는것과 같은 일정한 프로그람작성환경에서 쓸모있는 기억장치를 제공한다.

실례 7-8은 간단한 탄창클라스를 창조한다.

(실례 7-8) 탄창클라스

```
#include <iostream>
#include <iomanip>
using namespace std;
class Stack
{
```

```
private:
   enum { MAX = 10 };
   int st[MAX];
   int top;
public:
   Stack() { top = -1; }
   void Push(int var) { st[++top] = var; }
   int Pop() { return st[top--]; }
};
int main()
{
   Stack s1;
   s1.Push(11);
   s1.Push(22);
   cout << "1: " << s1.Pop() << endl;
   cout << "2: " << s1.Pop() << endl;
   s1.Push(33);
   s1.Push(44);
   s1.Push(55);
   s1.Push(66);
   cout << "3: " << s1.Pop() << endl;
   cout << "4: " << s1.Pop() << endl;
   cout << "5: " << s1.Pop() << endl;
   cout << "6: " << s1.Pop() << endl;
   return 0;
}
```

탄창의 중요한 성원은 배렬 st이다. int변수 top는 탄창에 보관된 마지막 항목의 첨수를 가리키고 항목은 탄창의 꼭대기에 배치된다. 탄창에 사용되는 배렬의 크기는 MAX에 의해 지정되고 MAX는 다음과 같이 정의된다.

```
enum[MAX=10];
```

MAX의 이러한 정의는 일반적인것이 아니다. 밀봉성을 보유하기 위하여 클라스안에서만 쓰이는 상수는 MAX와 같이 클라스안에서 정의하는것이 좋다. 이러한 목적에 const대역변수를 사용하는것은 최량이 아니다. 표준 C++에서는 클라스안의 MAX를 다음과 같이 즉

```
static const int MAX =10;
```

과 같이 선언할수 있어야 한다.

이것은 MAX가 상수이고 클라스의 모든 객체들에 적용될수 있다는것을 의미한다. 그러나 Microsoft Visual C++를 비롯한 일부 번역프로그람들은 이와 같은 구조를 허용하지 않는다.

이러한 상수를 렬거자로서 정의할수 있다. 이때 렬거이름은 요구되지 않고 오직 한개의 렬거자만 요구된다.

```
enum [MAX=10];
```

이것은 MAX를 값 10을 가진 옹근수로 정의하고 클라스안에 완전히 포함된다. 그

러나 이 수법은 제대로 동작하지만 불편하다. 번역프로그람이 static const수법을 보유하고있다면 클라스안에서 상수를 정의할 때 그것을 사용해야 한다.

그림 7-6은 탄창을 보여준다.

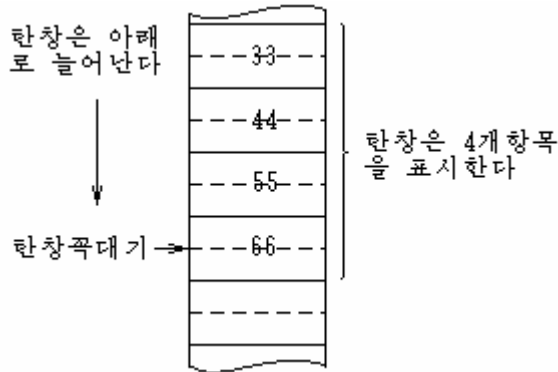



그림 7-6. 탄창

탄창에 항목을 밀어넣을 때 보관하려는 값을 인수로 가지는 성원함수 Push()를 호출한다. Pop()성원함수는 탄창에서 항목을 꺼내여 그 값을 돌려준다.

실례 7-8에서 main()프로그람은 Stack클라스를 사용하여 객체 s1을 창조하고 탄창에 두개의 항목을 밀어넣고 그것을 꺼내서 표시한다. 그다음 탄창에 네개의 더 많은 항목을 밀어넣고 그것들을 꺼내서 표시한다. 출력은 다음과 같다.

```
1: 22
2: 11
3: 66
4: 55
5: 44
6: 33
```

여기서 알수 있는것처럼 항목들을 탄창에 밀어넣고 꺼내며 마지막에 밀어넣은것은 처음에 얻어진다.

증가 및 감소연산자의 앞붙이와 뒤붙이표기를 사용한다. Push()성원함수에서 명령문

```
st[++top] =  var;
```

는 먼저 top를 증가시켜 다음의 유효배렬원소를 가리키게 한 다음 이 원소에 var를 대입하여 탄창의 새로운 꼭대기로 되게 한다. 명령문

```
return st[top--];
```

는 우선 탄창의 꼭대기에서 꺼낸 값을 돌려주고 top원소를 감소시켜 그 앞의 원소를 가리키게 한다.

탄창클라스는 용기 혹은 자료보관기구를 실현하는데 클라스를 사용하는 객체지향 프로그람작성법의 중요특성의 한가지 실례이다. 15장에서 탄창이 자료보관방법들의 하나라는것을 알수 있다. 자료보관방법에는 대기렬, 모임, 련결목록과 같은것이 있다. 자료보관방법은 프로그람의 고유한 요구에 맞는것을 선택한다. 자료보관방법을 제공하는

데 이미 존재하는 클라스를 사용하면 작성자가 자료보관기구의 세부를 복사하는데 시간을 소비할 필요가 없어진다.

제 5 절. 객체배렬

배렬을 객체에 포함하는 방법을 보았다. 또한 반대로 객체배렬을 창조할수도 있다. 여기서는 두가지 경우 즉 거리의 배렬과 카드배렬을 고찰한다.

1. 거리의 배렬

6장에서 새로운 자료형을 표시하는 객체로서 메터와 센치메터를 포함하는 Distance클라스의 실례를 여러개 보았다. 다음의 실례 7-9는 그러한 객체배렬을 보여준다.

(실례 7-9) Distance객체

```
#include <iostream>
using namespace std;
class Distance
{
private:
   int meters;
   float centies;
public:
   void GetDist()
   {
      cout << "\n  메터를 입력하십시오:"; cin >> meters;
      cout << "  센치메터를 입력하십시오:"; cin >> centies;
   }
   void ShowDist() const
      { cout << meters << "m " << centies << "cm"; }
};
int main()
{
   Distance dist[10];
   int n = 0;
   char ans;
   cout << endl;
   do
   {
      cout << n + 1 << "번째 거리를 입력하십시오" ;
      dist[n++].GetDist();
      cout << "계속하겠습니까(y/n)?: "; cin >> ans;
   } while(ans != 'n');
   for(int j=0; j<n; j++)
   {
```

```
        cout << endl << j + 1 << "번째 거리=";
        dist[j].ShowDist();
    }
    cout << endl;
    return 0;
}
```

이 프로그람에서 사용자는 필요한 량만큼 거리를 입력한다. 매개 거리를 입력한 다음 프로그람은 사용자에게 또 입력하겠는가를 묻는다. 아니면 완료하고 입력한 거리들을 모두 표시한다. 사용자가 세개의 거리를 입력했을 때의 대화가 있다.

```
1번째 거리를 입력하십시오
  메터를 입력하십시오:5
  센치메터를 입력하십시오:4
계속하겠습니까(y/n)?: y
2번째 거리를 입력하십시오
  메터를 입력하십시오:6
  센치메터를 입력하십시오:2.5
계속하겠습니까(y/n)?: y
3번째 거리를 입력하십시오
  메터를 입력하십시오:5
  센치메터를 입력하십시오:10.5
계속하겠습니까(y/n)?: n
1번째 거리=5m 4cm
2번째 거리=6m 2.5cm
3번째 거리=5m 10.5
```

물론 이미 입력한 거리를 그대로 표시할 대신에 프로그람은 그것들을 평균할수도 있고 디스크에 써넣거나 다른 방법으로 조작할수도 있다.

- 배렬의 한계

이 프로그람은 사용자로부터 입력을 받아들이는데 do순환을 사용한다. 이것은 사용자가 필요한 량만큼 즉 배렬크기의 한계 MAX까지 Part형구조체의 자료를 입력하게 하는 방법이다.

만일 사용자가 10개이상의 거리를 입력한다면 어떤 일이 생기는가?

그에 대한 대답은 미리 예측할수 없지만 확정적으로 나쁘다. C++배렬을 검사하는 한계는 없다. 프로그람이 배렬의 끝에 무엇인가 써넣는다면 번역프로그람이나 실시간체계객체는 그것을 알수 없다. 결국 자기의 자료를 변경하는것이 아니라 다른 자료의 우에 씌여지거나 혹은 프로그람코드자체에 씌여진다. 이것은 나쁜 효과를 일으키거나 체계를 완전히 중지시킨다.

중요한것은 작성자가 배렬한계를 검사하게 하는것이다. 배렬에 아주 많은 자료를 입력하여야 한다면 배렬은 사용자가 요구하는 크기보다 더 크게 만들어야 한다. 례를 들면 실례 7-9에서 do순환의 선두에 다음의 코드를 삽입할수 있다.

```
if(n >= MAX)
{
```

```
        cout << "\n배렬이 다 찼다!!!" ;
        break;
    }
```

이것은 순환을 중지하고 배렬의 자리넘침을 방지한다.

- 배렬안의 객체호출

이 프로그람에서 Distance클라스선언은 이전의 프로그람들에서와 비슷하다. 그러나 main()프로그람에서는 그러한 객체들의 배렬을 정의한다. 즉

```
Distance dist[MAX];
```

여기서 dist배렬의 자료형은 Distance이고 MAX개의 요소를 가진다. 그림 7-7은 이것을 보여준다.

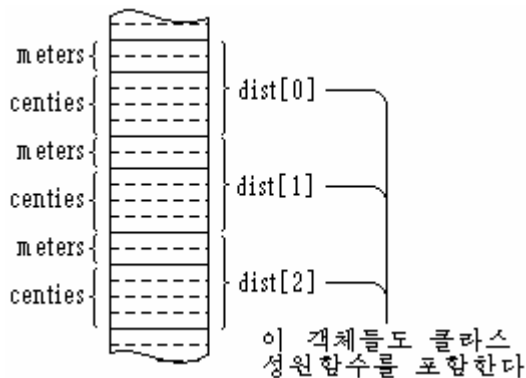



그림 7-7. 객체배렬

배렬원소의 클라스성원함수는 실례 7-7과 같이 배렬원소의 구조체성원과 류사하게 호출된다. 여기에 배렬 dist의 j번째 원소의 ShowDist()성원함수를 호출하는 방법이 있다.

```
dist[j].ShowDist();
```

보는바와 같이 배렬원소인 객체의 성원함수는 점연산자에 의하여 호출된다. 즉 점연산자에 의하여 중괄호안에 첨수가 있는 배렬의 이름과 소괄호가 있는 성원함수이름을 련결한다.

이것은 함수이름과 괄호가 자료이름대신에 사용되는것을 제외하면 구조체 혹은 클라스의 자료성원호출과 비슷하다.

배렬에 거리를 보관하기 위해 GetDist()성원함수를 호출할 때 배렬첨수 n을 증가시켜야 한다. 즉

```
dist[n++].GetDist();
```

이것은 사용자로부터 얻어진 자료의 다음 묶음을 dist안의 다음 배렬원소안의 구조체에 배치하는 방법이다. n변수는 do순환을 for대신 사용하므로 수동적으로 증가시켜야 한다. for순환에서 순환변수는 자동적으로 증가되고 배렬첨수로 쓰인다.

2. 카드배렬

더 큰 객체배렬의 실례를 고찰하자. 실례 6-10으로부터 Card클라스를 만들고 카

드 한조 모두의 배렬 deck를 창조해보자. 여기에 실례 7-10이 있다.

(실례 7-10) 카드배렬

```
#include <iostream>
#include <cstdlib>    // srand(), rand()
#include <ctime>
using namespace std;
enum Suit { clubs, diamonds, hearts, spades };
const int jack =  11;
const int queen = 12;
const int king =  13;
const int ace =   14;
class Card
{
private:
  int number;
  Suit suit;
public:
  Card() {}
  void Set(int n, Suit s) { number = n; suit = s; }
  void Display();
};
void Card::Display()
{
  if(number >= 2 && number <= 10)
     cout << number;
  else
  {
     switch(number)
     {
     case jack: cout << "J"; break;
     case queen: cout << "Q"; break;
     case king: cout << "K"; break;
     case ace: cout << "A"; break;
     }
  }
  switch(suit)
  {
  case clubs: cout << char(5); break;
  case diamonds: cout << char(4); break;
  case hearts: cout << char(3); break;
  case spades: cout << char(6); break;
  }
// cout << endl;
}
int main()
{
  Card deck[52];
```

```
    int j;
    cout << endl;
    for(j=0; j<52; j++) {
       int num = (j % 13) + 2;
       Suit su = Suit(j / 13);
       deck[j].Set(num, su);
    }
    cout << "\nOrdered deck:\n";
    for(j=0; j<52; j++) {
       deck[j].Display();
       cout << "  ";
       if(!((j+1) % 13))
           cout << endl;
    }
    srand(time(NULL));
    for(j=0; j<52; j++)
    {
       int k = rand() % 52;
       Card temp = deck[j];
       deck[j] = deck[k];
       deck[k] = temp;
    }
    cout << "\nShuffled deck:\n";
    for(j=0; j<52; j++)
    {
       deck[j].Display();
       cout << ", ";
       if(!((j+1) % 13))
           cout << endl;
    }
    return 0;
 }
```

deck안의 카드들을 표시하고 그것들을 뒤섞은 다음 다시 표시한다. 클라브, 다야몬드, 하트, 스페이드에 도형문자를 사용한다. 그림 7-8은 프로그람으로부터의 출력을 보여준다. 이 프로그람은 여러가지 새로운 개념들을 포함하므로 그것들을 차례로 고찰해보자.

```
Ordered deck:
2♧  3♧  4♧  5♧  6♧  7♧  8♧  9♧  10♧  J♧  Q♧  K♧  A♧
2♦  3♦  4♦  5♦  6♦  7♦  8♦  9♦  10♦  J♦  Q♦  K♦  A♦
2♥  3♥  4♥  5♥  6♥  7♥  8♥  9♥  10♥  J♥  Q♥  K♥  A♥
2♠  3♠  4♠  5♠  6♠  7♠  8♠  9♠  10♠  J♠  Q♠  K♠  A♠

Shuffled deck:
J♦, K♧, 2♥, 4♦, 5♧, 9♦, 10♠, 6♥, Q♧, 6♦, A♠, 9♧, A♦,
5♦, A♧, 2♠, 10♧, 3♠, 2♧, K♦, 9♠, 5♥, 8♦, 4♠, 7♥, 7♧,
3♧, 6♧, J♠, 7♠, 7♦, A♥, J♥, Q♦, 9♥, Q♥, 4♧, 8♧, 6♠,
K♠, 3♦, 5♠, 8♠, 3♥, K♥, 2♦, 8♥, 4♥, J♧, 10♦, Q♠, 10♥,
```

그림 7-8. 실례 7-10의 출력

- 도형기호

ASCII코드 32아래에 특수한 도형기호가 몇개 있다. Card의 Display()성원함수에서는 클라브, 다야몬드, 하트, 스페이드용 문자들을 호출하는데 ASCII코드 5, 4, 3, 6을 사용한다. 이 수를 char형으로 강제형변환하면 즉

```
static_cast<char>(5)
```

는 수값이 아니라 문자로서 <<연산자가 출력하게 한다.

- 카드배렬 deck

카드들의 묶음을 이루는 구조체배렬은 명령문 Card deck[52];에 의해 정의된다. 이 정의는 Card형객체들로 이루어지는 하나의 deck라는 배렬을 창조한다. deck안의 j번째 카드를 표시하기 위하여 Display()성원함수를 호출한다.

```
deck[j].Display();
```

- 우연수

우연수를 생성하여 사용할수 있다. 이 프로그람에서는 deck를 뒤섞는데 우연수들을 사용한다. 우연수를 얻는데 두개 단계가 요구된다. 우선 srand()서고함수를 호출하여 우연수발생기를 초기화한다. srand()함수는 기초우연수로서 체계시간을 사용하므로 두개의 머리부파일 CSTDLIB와 CTIME이 요구된다.

실제로 rand()서고함수를 호출하여 우연수를 생성한다. rand()함수는 우연옹근수를 돌려준다. 0～51범위의 수를 얻기 위해 rand()의 결과에 나머지연산자와 52를 적용한다.

```
int k = rand() % 52;
```

그다음 생성한 우연수를 두개의 카드로 교체하는데 사용한다. for순환을 통하여 0～51순서로 첨수가 가리키는 어떤 카드와 첨수가 우연수인 다른 카드를 서로 교체한다. 52개의 카드가 모두 우연카드로 바뀌여졌을 때 deck가 섞여진것으로 본다. 이 프로그람은 카드유희프로그람의 기초로 될수 있다.

객체배렬은 C++프로그람작성에 널리 쓰인다.

제 6 절. C문자렬

일반적으로 C++의 문자렬에는 두 종류 즉 C문자렬과 string클라스의 객체로서의 문자렬이 있다고 이미 언급하였다. 이 절에서는 첫종류의 문자렬을 론의한다.

이 문자렬은 C언어에서 유효한 문자렬의 유일한 종류이므로 C문자렬이라고 한다. 또한 char형에로의 지적자로 표시되므로 char*문자렬이라고도 한다. 문자렬을 string클라스를 사용하여 창조한다하더라도 많은 경우에 C문자렬이 우선시 되고있고 C문자렬은 여러가지 원인에 의하여 아직도 중요하다.

첫째로, C문자렬은 많은 C서고함수들에서 사용되고있다.

둘째로, 이전에 개발된 코드에서 계속 보게 된다.

셋째로, C++를 배우는 사람들에게 있어서 C문자렬은 아주 원시적이므로 기초를 리해하는데 편리하다.

1. C문자렬변수

다른 자료형처럼 문자렬은 변수 혹은 상수일수 있다. 복잡한 문자렬조작을 시험하기 전에 서로 다른 두가지 실체를 고찰한다. 여기에 하나의 문자렬변수를 정의하는 실례가 있다.

프로그람은 사용자에게 문자렬을 입력할것을 요구하고 문자렬변수에 입력한 문자렬을 보관한 다음 문자렬을 표시한다.

(실례 7-11) 간단한 문자렬변수

```
#include <iostream>
using namespace std;
int main()
{
    const int MAX = 80;
    char str[MAX];
    cout << "문자렬을 입력하시오: ";
    cin >> str;
    cout << "입력한 문자렬은 " << str << endl;
    return 0;
}
```

문자렬변수 str의 정의는 char형배렬의 정의와 같다.

```
char str[MAX];
```

발취연산자 >>를 사용하여 건반으로부터 문자렬을 읽어들이고 문자렬변수 str에 넣는다. 발취연산자는 문자렬이 문자들의 배렬이라는것과 문자렬을 취급하는 방법을 알고있다. 사용자가 프로그람에서 문자렬 "Computer"를 입력하면 배렬 str는 그림 7-9와 같이 기억기에 배치된다.

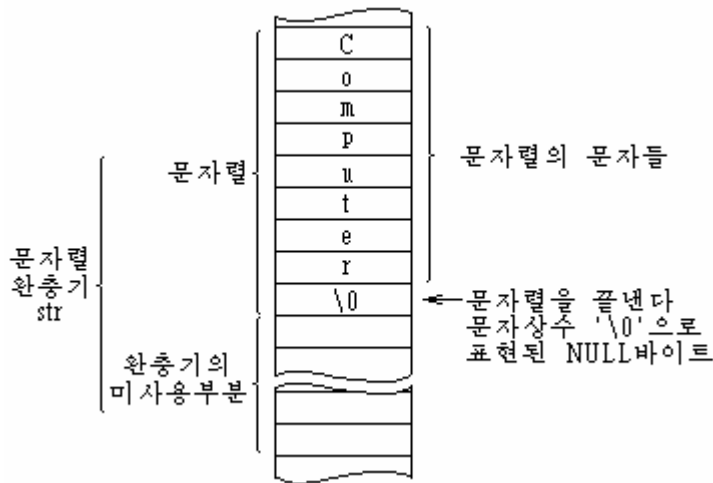



그림 7-9. 문자렬변수에 보관된 문자렬

매개 문자는 1byte의 기억기를 차지한다. C문자렬에서 중요한것은 0을 포함하는 바이트로 끝나야 하는것이다. 이것은 문자상수 '\0'으로 표시되고 이 문자는 0이라는 ASCII코드값을 가진다. '\0'으로 표시되는 문자를 null문자라고 한다. <<연산자가 문자렬을 표시할 때 null문자가 나타날 때까지 문자들을 표시한다.

2. 완충기넘침의 방지

실례 7-11은 사용자가 문자렬에 입력하게 한다.

그러면 사용자가 배렬의 길이보다 긴 문자렬을 입력하면 어떤 일이 생기는가?

이미 언급한것처럼 C++에는 배렬밖으로 벗어나는 배렬원소들의 삽입으로부터 프로그람을 보호하기 위한 기구가 없다. 그러나 >>연산자가 배렬안에 입력하는 문자수를 제한하게 할수 있다. 실례 7-12는 그 방법을 보여준다.

(실례 7-12) cin폭을 사용한 완충기넘침의 방지

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
   const int MAX = 20;
   char str[MAX];
   cout << "문자렬을 입력하시오: ";
   cin >> setw(MAX) >> str;
   cout << "입력한 문자렬은 " << str << endl;
   return 0;
}
```

이 프로그람은 setw조작자를 사용하여 입력완충기에 받아들일수 있는 최대문자수를 지정한다. 사용자가 문자들을 더 많이 입력하면 >>연산자는 그것들을 배렬에 모두

삽입하지 않는다. 실제로는 문자렬이 null문자로 끝나므로 지정된 수보다 하나 적은 문자가 삽입된다. 이리하여 실례 7-12에서 최대 19개의 문자가 삽입된다.

3. 문자렬상수

문자렬을 정의할 때 상수값으로 초기화할수 있다. 실례 7-13은 문자렬을 초기화한다.

(실례 7-13) 문자렬의 초기화

```
#include <iostream>
using namespace std;
int main()
{
    char str[] = "Computer Science.\n";
    cout << str << endl;
    return 0;
}
```

여기서는 문자렬상수를 인용부호에 넣어서 영어로 쓴다. 이것은 문자렬이 char형 배렬이므로 이상하게 생각할수 있다. 이전의 실례에서는 괄호에 의하여 구분되거나 반점으로 구분되는 일련의 값들로 배렬을 초기화하는것을 보아왔다.

그러면 왜 문자렬을 같은 방법으로 초기화하지 않는가?

사실 그러한 문자상수들의 렬을 사용할수 있다. 실례로

```
char str[] = { 'C', 'o', 'm', 'p', 'u', … };
```

C++의 설계가들은 실례 7-13과 같은 간단한 수법을 제공해준다. 그 효과는 같다. 문자들은 배렬에 차례로 보관되고 C문자렬처럼 마지막 문자는 null이다.

4. 공백을 포함하는 문자렬의 읽기

한개이상의 단어를 포함하는 문자렬을 가지고있는 실례 7-11의 프로그람을 시험해보면 이상하게 생각할수 있다. 여기에 그 실례가 있다.

```
문자렬을 입력하시오: Computer Science
입력한 문자렬은 Computer
```

문장의 나머지는 어디로 갔는가? 발취연산자 >>는 공백을 마감문자로 간주하므로 한단어의 문자렬을 읽어들이며 공백뒤에 입력한 문자렬은 없어진다.

함수 cin::get()를 사용하여 공백을 포함하는 본문을 읽을수 있다. 이것은 cin이 객체로 되여있는 스트림클라스의 성원함수 get()를 의미한다. 실례 7-14에 그 사용방법을 보여준다.

(실례 7-14) 공백을 포함하는 문자렬읽기

```
#include <iostream>
using namespace std;
int main()
{
```

```
    const int MAX = 80;
    char str[MAX];
    cout << "문자렬을 입력하시오: ";
    cin.get(str, MAX);
    cout << "입력한 문자렬은 " << str << endl;
    return 0;
  }
```

cin::get()의 첫 인수는 입력하는 문자렬을 보관하는 배렬주소이다. 둘째인수는 배렬의 최대크기를 지정한다. 이리하여 자동적으로 완충기를 벗어나는 실행을 피하고있다.

이 함수를 사용하면 입력흐름이 문자렬의 실체안에 보관된다.

```
문자렬을 입력하시오: Computer Science
입력한 문자렬은 Computer Science
```

cin.get()와 발취연산자(>>)를 결합할 때 잠재적인 문제가 있다. 12장에서는 이 문제를 해결하기 위하여 cin의 ignore()성원함수를 론의한다.

5. 여러행의 읽기

공백이 들어있는 문자렬을 읽는 문제를 해결하였다. 그러면 여러행을 가지는 문자렬을 어떻게 읽겠는가? cin::get()함수는 이러한 경우에 제3인수를 사용할수 있다. 세번째 인수는 읽기를 중지하도록 함수에게 알리는 문자를 지정한다.

인수의 기정값은 새행('\n')문자이지만 다른 문자를 지정하여 함수를 호출할수도 있다. 이때 기정값이 주어진 문자로 교체된다.

다음의 실례에서는 '$'기호를 제3인수로 하여 함수를 호출한다.

(실례 7-15) '$'문자로 끝나는 여러행 읽기

```
#include <iostream>
using namespace std;
const int MAX = 2000;
char str[MAX];
int main()
{
  cout << "\n문자렬을 입력하시오:\n";
  cin.get(str, MAX, '$');
  cout << "입력한 문자렬은\n" << str << endl;
  return 0;
}
```

이제는 필요한 량만큼 여러행을 입력할수 있다. get()함수는 마감문자를 입력하거나 배렬크기를 넘을 때까지 계속 문자들을 받아들인다. '$'문자를 입력한 다음 Enter건을 눌러야 한다. 여기에 리수복영웅의 시의 여러구절을 입력했을 때의 출력이 있다.

```
문자렬을 입력하시오:
리수복영웅의 시
나는 해방된 조선의 청년이다
```

```
$
입력한 문자렬은
리수복영웅의 시
나는 해방된 조선의 청년이다
```

한행은 Enter로 끝나지만 프로그람은 '$'문자를 입력할 때까지 입력을 계속 받아들인다.

6. 문자렬복사

- 문자렬을 복사하는 고전적인 방법

문자렬의 본질을 리해하는 가장 좋은 방법은 그것들을 문자로 취급하는것이다. 다음의 실례에서 그렇게 한다.

(실례 7-16) for순환을 사용한 문자렬복사

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
   char str1[] = "나는 평양철도대학 철도운영학부 "
      "정보공학과 학생이다";
   const int MAX = 80;
   char str2[MAX];
   for(int j=0; j < strlen(str1); j++)
      str2[j] = str1[j];
   str2[j] = '\0';
   cout << str2 << endl;
   return 0;
}
```

이 프로그람은 문자렬변수 str1과 문자렬변수 str2를 창조한 다음 for순환에 의하여 문자렬상수를 문자렬변수에 복사한다. 복사는 한번에 한문자씩 진행된다. 즉

```
str2[j] = str1[j];
```

번역프로그람은 두개의 이웃 문자렬들을 하나의 문자렬에 결합한다. 이것은 두행에 인용문을 갈라 쓸수 있게 한다.

실례에서는 또한 C문자렬서고함수를 소개한다. C++의 내부에는 문자렬연산자가 없으므로 C문자렬은 보통 서고함수에 의하여 조작되여야 한다. 그러한 함수는 많다. 실례에서 사용하는 strlen()은 C문자렬의 길이를 얻는다. 이 길이는 정확한 문자수가 복사되도록 for순환을 제한하는 값으로서 사용된다. 문자렬함수를 사용할 때 머리부파일 CSTRING을 포함하여야 한다.

문자렬의 복사판은 null로 끝나야 한다. 그러나 strlen()에 의해 돌아온 문자렬길이에는 null이 포함되지 않는다. 하나의 추가문자를 복사할수 있으나 null을 명백히 삽입하는것이 더 안전하다. 다음 행에서 그것을 수행한다.

```
srt2[j] = '\0';
```

이 문자를 삽입하지 않으면 프로그람에 의해 표시된 문자렬에 이상한 문자들의 렬이 포함되여있는것을 볼수 있다. <<는 '\0'을 만날 때까지 모든 인쇄문자들을 출력한다.

- 문자렬을 복사하는 간단한 방법

물론 문자렬복사에 for순환을 사용할 필요는 없다. 우의 조작을 서고함수가 수행해준다. 여기에 실례 7-17이 있다. 여기서는 strcpy()함수를 사용한다.

(실례 7-17) strcpy()함수를 사용한 문자렬복사

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
   char str1[] = "나는 평양철도대학 운영학부 "
      "정보공학과 학생이다";
   const int MAX = 80;
   char str2[MAX];
   strcpy(str2, str1);
   cout << str2 << endl;
   return 0;
}
```

다음과 같이 함수를 호출할수 있다.

```
strcpy(<목적문자렬>, <원천문자렬>);
```

오른쪽에 있는 변수는 왼쪽에 있는 변수에 복사된다.

7. 문자렬배렬

배렬들의 배렬이 있듯이 문자렬들의 배렬도 있다. 실례 7-18은 배렬에 요일들의 이름을 보관한다.

(실례 7-18) 문자렬배렬

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
   const int DAYS = 7;
   const int MAX = 10;
   char star[DAYS][MAX] = { "Sunday", "Monday", "Tuesday",
      "Wednesday", "Thirsday", "Friday", "Saturday" };
   for(int j=0; j<DAYS; j++)
      cout << star[j] << endl;
   return 0;
}
```

프로그람은 배렬로부터 매개 문자렬을 출력한다.

```
Sunday
Monday
```

```
Tuesday
Wednesday
Thirsday
Friday
Saturday
```

문자렬들의 배렬이므로 문자렬배렬 star는 실제로 2차원배렬이다. 이 배렬의 1차원크기 DAYS는 배렬에 몇개의 문자렬이 있는가를 지정한다. 2차원크기 MAX는 문자렬들의 최대길이를 지정한다. 그림 7-10은 이것을 보여준다.

10

7		0	1	2	3	4	5	6	7	8	9	
	0	S	u	n	d	a	y	\0				star[0]
	1	M	o	n	d	a	y	\0				star[1]
	2	T	u	e	s	d	a	y	\0			star[2]
	3	W	e	d	n	e	s	d	a	y	\0	star[3]
	4	T	h	i	r	s	d	a	y	\0		star[4]
	5	F	r	i	d	a	y	\0				star[5]
	6	S	a	t	u	r	d	a	y	\0		star[6]

그림 7-10. 문자렬배렬

최대길이보다 작은 문자렬들에서는 여러개의 바이트가 랑비된다. 지적자를 설명할 때 이와 같은 비효과성을 제거하는 방법을 설명한다.

개별적인 문자렬을 호출하는 문법은 다음과 같다.

```
star[j];
```

문자렬배렬을 2차원배렬로서 론한다면 둘째 첨수는 어디에 있는가?

2차원배렬은 배렬들의 배렬이므로 그 매개가 배렬(즉 이 실례에서는 문자렬의 밖에 있는 배렬)의 원소들을 호출할수 있다. 여기서는 둘째 첨수가 필요없다. 따라서 star[j]는 문자렬들의 배렬에서 문자렬번호 j이다.

8. 클라스성원으로서 배렬

문자렬은 자주 클라스의 성원으로 사용된다. 실례 6-2를 변경한 실례 7-19는 C문자렬을 부분품의 이름을 보관하는데 사용한다.

(실례 7-19) 부분품객체에서 문자렬의 사용

```
#include <iostream>
#include <cstring>
using namespace std;
class Part
{
private:
   char partName[30];
   int partNumber;
   float cost;
public:
```

```
    void SetPart(char pName[], int pn, char c)
    {
        strcpy(partName, pName);
        partNumber = pn;
        cost = c;
    }
    void ShowPart()
    {
        cout << "\n  이름=" << partName;
        cout << ", 번호=" << partNumber;
        cout << ", 단가=" << cost << "원";
    }
};
int main()
{
    Part part1, part2;
    part1.SetPart("볼트", 4473, 217.55);
    part2.SetPart("나트", 9924, 419.25);
    cout << "\n첫째 부분품: "; part1.ShowPart();
    cout << "\n둘째 부분품: "; part1.ShowPart();
    cout << endl;
    return 0;
}
```

이 프로그람은 Part클라스의 두번째 객체를 정의하고 SetPart()성원함수에 의하여 거기에 값을 준다. 그다음 ShowPart()성원함수에 의하여 그것들을 표시한다. 실행결과는 다음과 같다.

```
첫째 부분품:
  이름=볼트, 번호=4473, 단가=217.55원
둘째 부분품:
  이름=나트, 번호=9924, 단가=419.25원
```

프로그람의 크기를 줄이기 위하여 클라스성원들에서 형번호를 생략한다.

SetPart()성원함수에서는 strcpy()문자렬서고함수에 의하여 인수 pName으로부터 클라스자료성원 partName에 문자렬을 복사한다. strcpy()함수는 대입문이 기본형의 변수들에 대하여 수행하는것과 같은 조작을 수행한다.

이밖에 문자렬추가, 문자렬비교, 문자렬에서 지정문자의 검색 등 많은 서고함수가 있다.

9. 사용자정의문자렬형

C문자렬을 C++에서 사용할 때 일련의 문제가 제기된다. 그 하나는 대입식을 사용할수 없는것이다. 즉

```
strDest = strSrc;
```

표준 C++ string클라스가 이 문제를 해결해주지만 객체지향기술을 사용하여 자체로 문제를 해결할수도 있다.

자체의 문자렬클라스를 창조하면 문자렬을 클라스의 객체로 포함할수 있다.

C++클라스를 사용하여 자체로 String형을 정의한다면 대입명령문을 사용할수 있다.

(실례 7-20) 문자렬클라스

```
#include <iostream>
#include <cstring>
using namespace std;
class String
{
private:
   enum { SZ=80 };
   char str[SZ];
public:
   String() { str[0] = '\0'; }
   String(char s[]) { strcpy(str, s); }
   void Display() { cout << str; }
   void ConCat(String s2)
   {
      if(strlen(str) + strlen(s2.str) < SZ)
          strcat(str, s2.str);
      else
          cout << "\n문자렬이 너무 깁니다.";
   }
};
int main()
{
   String s1("평양철도대학 ");
   String s2 = "정보공학과";
   String s3;
   cout << "\ns1="; s1.Display();
   cout << "\ns2="; s2.Display();
   cout << "\ns3="; s3.Display();
   s3 = s1;
   cout << "\ns3="; s3.Display();
   s3.ConCat(s2);
   cout << "\ns3="; s3.Display(); cout << endl;
   return 0;
}
```

String클라스는 char형배렬을 포함한다. 새로 정의한 클라스는 문자렬의 원시정의 즉 char형배렬과 같아보이게 한다. 그러나 클라스안에 배렬을 포함하였으므로 일련의 우점이 얻어진다. 대입연산자에 의하여 한 객체를 같은 클라스의 다른 객체에 대입할수 있으므로 s3 = s1;과 같은 명령문을 쓸수 있다. 이것은 한 문자렬객체를 다른 문자렬객체와 같게 설정한다. 또한 String을 취급하는 자체의 성원함수를 정의할수 있다.

실례 7-20에서는 모든 String들이 같은 길이 즉 SZ문자를 가진다.

구성자가 두개 있다. 첫 구성자는 str의 첫 문자를 null문자로 설정하므로 문자렬은 길이 0을 가진다. 이 구성자는 String s3;과 같은 명령문에서 호출된다.

둘째 구성자는 String객체를 표준문자렬상수(C문자렬)로 설정한다. 문자렬상수를 자기의 자료에 복사하는데 strcpy()서고함수를 사용한다. 즉

```
String s1("평양철도대학 ");
```

이 구성자는 1인수구성자이다.

```
String s2 = "정보공학과";
```

어느 형식을 사용하든 이 구성자는 C문자렬을 String으로 효과적으로 변환한다. 즉 표준문자렬상수를 String클라스의 객체로 변환한다. 성원함수 Display()는 String을 표시한다.

String의 다른 성원함수 ConCat()는 한 String이 다른 String과 작업한다. 원시 String은 ConCat()가 성원인 객체이다. 이 String에 인수로 넘어온 String이 추가된다. 따라서 main()의 명령문

```
s3.ConCat(s2);
```

은 s2을 s3에 추가한다. s2이 "정보공학과"로 초기화되고 s3에는 s1의 값 "평양철도대학 "이 할당되였으므로 s3의 결과값은 "평양철도대학 정보공학과"로 된다.

ConCat()함수는 문자렬련결에 strcat()서고함수를 사용한다. strcat()는 첫 인수로 지정된 문자렬에 둘째 인수로 지정된 문자렬을 추가한다. 프로그람의 출력은 다음과 같다.

```
s1=평양철도대학
s2=정보공학과
s3=  ←아무것도 없다
s3=평양철도대학  ←s1과 같아진다
s3=평양철도대학 정보공학과
```

ConCat()함수에 주어진 둘째 String이 String길이의 최대값을 초과하면 그 문자렬이 련결되지 않고 사용자에게 통보문이 출력된다.

제 7 절. 표준 C++ string클라스

표준 C++는 string이라는 새로운 클라스를 포함한다. 이 클라스는 전통적인 C문자렬에 비하여 많은 개선을 준다. 그 하나는 문자렬변수를 보관하는 정확한 크기의 배렬을 창조할수 있는것이다. string클라스는 기억관리를 자체로 부담한다. 또한 string클라스에서는 재정의된 연산자를 사용하므로 +연산자에 의한 문자렬의 결합이 기본이다.

물론 다른 우점도 있다. string클라스는 C문자렬보다 효과적이고 안전하다. (그러나 C문자렬을 써야 하는 경우가 아직 많이 남아있다.) 이 절에서는 string클라스와 각종 성원함수와 연산자들을 시험한다.

1. string객체의 정의와 대입

여러가지 방법으로 string객체를 정의할수 있다. 인수없는 구성자에 의하여 빈문자렬을 창조할수 있다. 또한 1인수구성자를 사용할수 있다. 여기서 인수는 C문자렬상수 즉 2중인용기호에 의해 구분된 문자들이다. 클라스 string의 객체는 단순대입연산자를 사용하여 서로 대입할수 있다. 아래에 실례 7-21이 있다.

(실례 7-21) string객체의 정의와 대입

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string s1("평양철도대학 ");
    string s2 = "학생";
    string s3;
    s3 = s1;
    cout << "s3=" << s3 << endl;
    s3 = "나는 " + s1 + "정보공학과";
    s3 += s2;
    cout << "\ns3=" << s3 << endl;
    s1.swap(s2);
    cout << s1 << " 철도운영학부 " << s2 << endl;
    return 0;
}
```

코드의 처음 세개 행은 string객체의 정의방법을 보여준다. 처음 두개는 string들을 초기화하고 셋째는 빈 string변수를 창조한다. 다음 행은 대입연산자에 의한 단순대입을 보여준다.

string클라스에는 재정의된 연산자들이 많다. 다음 장에서 연산자재정의에 대하여 설명하지만 그러한 지식이 없어도 이 연산자를 사용할수 있다.

재정의된 +연산자는 한 문자렬객체를 다른 문자렬객체와 련결한다. 명령문

```
s3 = "나는 " + s1 + "정보공학과";
```

는 변수 s3에 문자렬 "나는 평양철도대학 정보공학과"를 보관한다.

또한 현존문자렬의 끝에 문자렬을 추가하는데 +=연산자를 사용할수 있다. 명령문

```
s3 += s2;
```

은 "학생"인 s2을 s3의 끝에 추가하여 문자렬 "나는 평양철도대학 정보공학과 학생"을 창조하고 그것을 s3에 대입한다.

또한 이 실례는 두개 문자렬객체들의 값을 교환하는 string클라스의 성원함수 swap()를 소개한다. swap()는 어떤 객체가 다른 객체를 인수로 하여 호출한다.

그것을 s1("평양철도대학 ")과 s2("학생")에 적용하고 현재 s1이 자료 "학생", s2이 "평양철도대학 "이라는것을 보여주기 위하여 그 값들을 표시한다.

실례 7-21의 출력은 다음과 같다.

```
s3=평양철도대학
s3=나는 평양철도대학 정보공학과
s3=나는 평양철도대학 정보공학과 학생
학생 철도운영학부 평양철도대학
```

2. string객체를 사용한 입력과 출력

입출력은 C문자렬과 같은 방법으로 조종된다. <<와 >>연산자들은 string객체들을 조종할수 있도록 재정의되여있다. 그리고 함수 getline()은 공백을 포함하거나 여러행을 포함하는 입력을 조종한다. 실례 7-22에 이것을 보여준다.

(실례 7-22) string클라스의 입출력

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string subjName, name, address;
   string greeting("안녕하십니까!, ");
   cout << "학부학과이름을 입력하시오: ";
   getline(cin, subjName);
   cout << "학부학과이름은 " << subjName << endl;
   cout << "이름을 입력하시오: ";
   cin >> name;
   greeting += name;
   cout << greeting << endl;
   cout << "주소를 입력하시오\n";
   cout << "'$'문자로 끝내시오\n";
   getline(cin, address, '$');
   cout << "주소는 " << address << endl;
   return 0;
}
```

프로그람은 공백이 들어있는 사용자의 학부학과이름을 getline()에 의하여 읽어들인다. getline()함수는 C문자렬에 사용된 get()함수와 비슷하지만 성원함수가 아니다. 그대신 그 첫 인수가 입력스트림객체이다. 둘째 인수는 본문이 보관되는 string객체 즉 subjName이다. 이 변수는 그다음 cout와 <<에 의하여 표시된다.

다음으로 프로그람은 한개 단어로 이루어지는 사용자의 이름을 cin과 >>연산자에 의해 읽어들인다. 끝으로 프로그람은 세개의 인수를 가지는 getline()에 의하여 사용자의 주소(여러행)를 읽어들인다. 셋째 인수는 입력을 끝내는데 쓰이는 문자를 지정한다. 이 프로그람에서는 '$'기호를 사용한다.

사용자는 Enter건을 누르기전에 마지막인수로서 이 문자를 입력해야 한다. 세번째 인수가 없으면 getline()의 구분기호는 '\n'으로 된다. 대화는 다음과 같다.

```
학부학과이름을 입력하시오: 철도운영학부 정보공학과
학부학과이름은 철도운영학부 정보공학과
```

```
이름을 입력하시오: 김철호
안녕하십니까!,김철호
주소를 입력하시오
'$'문자로 끝내시오
만경대구역 칠골 3동 1반$
주소는 만경대구역 칠골 3동 1반
```

3. string객체의 검색

string클라스에는 string객체안의 문자렬과 보조문자렬들을 검색하는 성원함수들이 포함되여있다. 실례 7-23에서 그것을 보여준다.

(실례 7-23) string객체에서 보조문자렬검색

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string s1 = "abc bcd cde def efg cde";
   int n;
   n = s1.find("bcd");
   cout << n << "에서 bcd를 발견하였습니다." << endl;
   n = s1.find_first_of("cde");
   cout << n << "에서 'c','d','e'중 하나를 처음으로"
              "발견하였습니다." << endl;
   n = s1.find_first_not_of("aioue");
   cout << n << "에서 첫째 자음을 발견하였습니다." << endl;
   return 0;
}
```

find()함수는 그것을 호출한 문자렬안에서 인수로서 넘어온 문자렬을 찾는다. 여기서는 s1에서 "bcd"를 찾는다. s1에는 "bcd cde"의 행이 들어있다. 따라서 find()는 위치 4에서 "bcd"를 찾는다. C문자렬처럼 제일 왼쪽 문자위치는 0이다.

find_first_of()함수는 문자들의 묶음에 들어있는 임의의 문자를 찾고 발견한 첫 문자의 위치를 돌려준다. 여기서는 'c', 'd', 'e'중에서 임의의 문자를 찾는다. 처음 발견하는 문자는 "abc"의 'c'(위치2)이다.

류사한 함수 find_first_not_of()는 지정된 묶음에 포함되지 않은 첫 문자를 그 문자렬에서 검색한다. 여기서는 묶음이 모두 모음으로 이루어져있고 대소문자가 모두 포함되므로 함수는 첫 자음 즉 둘째 문자를 검색한다. 실례 7-23의 출력은 다음과 같다.

```
4에서 bcd를 발견하였습니다.
2에서 'c','d','e'중 하나를 처음으로 발견하였습니다.
1에서 첫째 자음을 발견하였습니다.
```

이와 같은 함수들에는 여러 종류가 있으나 그것들은 고찰하지 않는다. 즉 rfind()는 문자렬을 역방향으로 조사한다. find_last_of()는 문자묶음의 어느하나와 일치하는 마지막 문자를 검색한다.

4. string객체의 변경

string객체를 변경하는 방법에는 여러가지가 있다. 다음의 실례는 성원함수 erase(), replace(), insert()를 보여준다.

(실례 7-24) string객체의 일부를 변경

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string s1("abc bcd cde def efg.");
   string s2("xyz");
   string s3("klm");
   s1.erase(0, 4);
   s1.replace(4, 3, s2);
   s1.replace(10, 1, "s");
   s1.insert(0, s3);
   s1.erase(s1.size() - 1, 1);
   int x = s1.find(' ');
   while(x < s1.size())
   {
      s1.replace(x, 1, "/");
      x = s1.find(' ');
   }
   cout << "s1=" << s1 << endl;
   return 0;
}
```

erase()함수는 string에서 부분문자렬을 삭제한다. 첫 인수는 부분문자렬안의 첫 문자의 위치이고 둘째 인수는 부분문자렬의 길이이다. 실례에서는 문자렬의 선두에서 "abc"를 지운다. replace()함수는 문자렬의 일부를 다른 문자렬과 바꾼다. 첫 인수는 교체가 시작되는 위치이고 둘째 인수는 교체하려는 원시문자렬의 문자수, 셋째 인수는 교체문자렬이다. 여기서는 "cde"를 "xyz"와 교체한다.

insert()함수는 첫 인수에 의해 지정된 위치에 둘째 인수에 의해 지정된 문자렬을 삽입한다. 여기서는 s1의 선두에 "klm"을 삽입한다. erase()를 두번째로 사용할 때 string객체안의 문자수를 돌려주는 size()성원함수를 사용한다. 식 size()-1은 마지막 문자의 위치이다. 이 문자는 삭제된다. append()함수는 문장의 끝에 세개의 물음표를 추가한다. 이 함수에서 첫 인수는 추가하려는 문자수이고 둘째는 추가하려는 문자이다.

프로그람의 끝에서는 부분문자렬의 여러개의 실례를 다른 문자렬과 교체하는 방법을 보여준다. 여기서 while순환에서는 find()를 사용하여 공백문자 ' '를 찾고 그것을 replace()에 의하여 사선과 교체한다.

문자렬 "abc bcd cde def efg."를 포함하는 s1로 시작한다.

변경후의 출력은 다음과 같다.

```
s1=klmbcd/xyz/sef/efg!!!
```

5. string객체들의 비교

재정의된 연산자나 compare()에 의하여 문자렬객체들을 비교할수 있다. compare()는 문자렬이 같은가, 그것들이 서로 자모순으로 앞서는가, 뒤에 놓이는가 등을 비교한다. 실례 7-25는 그 일부 가능성을 보여준다.

(실례 7-25) string객체의 비교

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string aName = "KimYongChol";
   string userName;
   cout << "Enter your name: ";
   cin >> userName;
   if(userName == aName)
      cout << "Greeetings, KimYongChol!\n";
   else if(userName < aName)
      cout << "You come before KimYongChol.\n";
   else
      cout << "You come after KimYongChol.\n";
   int n = userName.compare(0, 3, aName, 0, 3);
   cout << "The first three letters of your name ";
   if(n == 0)
      cout << "match ";
   else if(n < 0)
      "come before ";
   else
      cout << "come after ";
   cout << aName.substr(0, 3) << endl;
   return 0;
}
```

프로그람의 첫 부분에서 ==와 <연산자에 의하여 "KimYongChol"이라는 이름과 사용자가 입력한 이름이 같은가, 자모순으로 앞서는가, 뒤에 놓이는가를 결정한다. 프로그람의 둘째 부분에서 compare()함수는 "KimYongChol"의 처음 세개 문자와 사용자가 입력한 이름의 처음 세개 문자를 비교한다. compare()의 인수들에서는 userName에서 시작하거나 비교하려는 문자들의 수, 비교에 사용된 문자렬(aName), aName에서 시작위치와 문자수이다. 아래에 프로그람과의 대화가 있다.

```
Enter your name: AnChangHo
You come before KimYongChol.\n";
The first three letters of your name come before Kim
```

"KimYongChol"에서 처음 세개 문자는 substr()성원함수에 의하여 얻는다. substr()는 호출된 문자렬의 부분문자렬을 보내온다. 첫 인수는 부분문자렬의 위치, 둘째 인수는 문자수이다.

6. string객체의 문자호출

string객체안의 개별적인 문자들을 여러가지 방법으로 호출할수 있다. 다음의 실례에서는 at()성원함수에 의하여 호출한다. 또한 재정의된 []연산자를 사용하여 string객체를 배렬처럼 취급한다. 그러나 []연산자는 한계밖에 있는 문자를 호출하려고 시도하는 경우에 오유를 경고하지 않는다. []연산자는 실제배렬처럼 동작하고 더 효과있다. 그러나 진단하기 어려운 프로그람오유를 일으킬수 있다. 한계밖의 첨수를 사용하면 프로그람을 중지시키는 at()함수를 사용하는것이 더 안전하다.(14장 참고)

(실례 7-26) string객체의 문자호출

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string chArray;
   string word;
   cout << "Enter a word: ";
   cin >> word;
   int wLen = word.length();
   cout << "One character at a time:";
   for(int j=0; j<wLen; j++)
      cout << word.at(j);
//cout << word[j];
//word.copy(chArray, wLen, 0);
//chArray[wLen] = 0;
   chArray = word;
   cout << "\nArray containes: " << chArray << endl;
   return 0;
}
```

이 프로그람에서는 at()를 사용하여 string객체안의 문자들을 모두 표시한다. at()의 인수는 문자렬안의 문자위치이다.

그다음 copy()성원함수를 사용하여 string객체를 char형배렬의 C문자렬로 객체를 변환하면서 복사한다. 복사에 이어 배렬의 마지막 문자뒤에 null문자('\0')를 삽입하여 C문자렬에로의 변환을 끝마친다. string의 length()성원함수는 size()와 같은 수를 돌려준다. 프로그람의 출력은 다음과 같다.

```
Enter a word: Symbolic
One character at a time: Symbolic
Array containes: Symbolic
```

c_str() 또는 data()성원함수를 사용하여 string객체를 C문자렬로 변환할수 있다. 그러나 이 함수를 사용하자면 지적자를 알아야 한다.

7. string의 기타 함수들

size()와 length()가 둘다 string객체안의 문자수를 돌려주는것을 보았다. 일반적으로 문자렬이 차지하는 기억기의 량은 실제로 문자렬에 필요한것보다 더 많다.

capacity()성원함수는 실제로 차지한 기억기의 크기를 돌려준다. 이 한계에 이를때까지는 기억기를 확장하지 않고 문자렬에 문자들을 추가할수 있다. max_size()성원함수는 최대로 가능한 문자렬객체의 크기를 돌려준다. 이 크기는 작성자의 체계에서 int변수의 크기와 일치한다. 32bit Windows체계에서 이 값은 4,294,967,293byte이지만 기억용량은 대체로 이 량에 이르지 못한다.

string의 성원함수에는 인수의 개수와 형에 따라서 여러가지 종류가 있다.

string객체는 C문자렬처럼 null이나 0으로 끝나지 않는다. 그대신 문자렬의 길이는 클라스의 성원이다. 그러므로 문자렬을 순환할 때 끝에 이르렀는가를 알아보기 위하여 null을 검사해서는 안된다.

실제로 string클라스는 형판클라스 basic_string으로부터 파생된 수많은 문자렬형 클라스들중의 하나이다. string클라스는 char형에 기초하지만 다른 종류로서 wchar_t형을 사용할수 있다. 이것은 영어보다 더 많은 문자를 가지고있는 나라들의 언어에 basic_string을 사용할수 있게 한다. 번역프로그람의 방조파일은 basic_string아래에 있는 string의 성원함수들을 표시할수 있다.

요 약

배렬은 같은 형의 자료를 많이 포함한다. 자료의 형은 기본자료형, 구조체, 클라스일수 있다. 배렬안의 항목들을 원소라고 한다. 원소는 수값에 의해 호출되는데 이 수를 첨수라고 한다. 원소는 배렬이 정의될 때 특정값으로 초기화된다.

배렬은 여러 차원을 가질수 있다. 2차원배렬은 배렬들의 배렬이다. 배렬의 주소를 함수에서 인수로서 사용할수 있다. 이때 배렬자체는 복사되지 않는다. 클라스안에서 배렬을 성원자료로서 사용할수 있다. 그리고 자료가 배렬의 밖에 있는 기억기에 놓이지 않도록 주의해야 한다.

C문자렬은 char형의 배렬이다. C문자렬에서 마지막 문자는 null문자 '\0'이다. C문자렬상수는 특별한 형식을 가진다. 본문은 2중인용표에 넣는다. C문자렬조작에 여러가지 서고함수를 사용한다. C문자렬과 배렬은 char형배렬들의 배렬이다. C문자렬변수를 창조할 때 배렬이 임의의 본문을 보관하는데 충분하도록 담보해야 한다. C문자렬은 C

형서고함수들에 인수로서 쓰이며 이전 프로그람들에서 많이 볼수 있다. 일반적으로 새로운 프로그람들에서는 C문자렬을 사용하지 말아야 한다.

우수한 문자렬조작방법은 string클라스의 객체를 사용하는것이다. 여러가지 재정의된 연산자와 성원함수들에 의하여 string문자렬을 조작할수 있다. 사용자는 string객체들에서 기억관리에 대하여 근심할 필요가 없다.

문 제

1. 배렬원소는

① FIFO(먼저 넣은것을 먼저 꺼내기)수법

② 점연산자

③ 성원이름

④ 첨수

를 사용하여 호출한다. 어느것이 옳은가?

2. 배렬의 모든 원소들은 어떤 자료형이여야 하는가?

3. 100개 원소를 가지는 double형의 1차원배렬 doubleArray를 정의하는 명령문을 쓰시오.

4. 10개원소를 가지는 배렬은 첨수번호로서 어떤 범위의 수를 가지는가?

5. 배렬 doubleArray의 원소 j를 cout와 삽입연산자를 사용하여 출력하는 명령문을 쓰시오.

6. 원소 doubleArray[7]은 배렬에서

① 6번째 원소

② 7번째 원소

③ 8번째 원소

라고 말할수 있다. 어느것이 옳은가?

7. int형의 배렬 coins를 정의하고 1, 5, 10, 25, 50, 100으로 초기화하시오.

8. 다차원배렬을 호출할 때 매개 배렬원소는

① 반점으로 구분된다.

② 괄호안에 넣은 반점에 의해 구분된다.

③ 반점으로 구분되고 괄호에 둘러싸인다.

④ 괄호안에 넣는다.

어느것이 옳은가?

9. twoD라는 2차원배렬에서 보조배렬 2의 원소 4를 호출하는 식을 쓰시오.

10. C++에 다차원배렬이 있는가?

11. float형의 2차원배렬 flArr에 대하여 배렬을 선언하고 첫째 보조배렬을 52, 27,

83, 둘째 보조배렬을 94, 73, 49로, 셋째 보조배렬을 3, 6, 1로 초기화하는 식을 쓰시오.

12. 원천파일에 쓰는 배렬이름은 무엇을 표시하는가?

13. 배렬이름을 함수에 넘길 때 함수는

① 호출측프로그람과 같은 배렬을 정확히 호출한다.

② 프로그람이 넘긴 배렬의 사본을 호출한다.

③ 호출측프로그람에 사용된것과 같은 이름을 사용하는 배렬을 참고한다.

④ 호출측프로그람에 의해 사용된것과 다른 이름을 사용하는 배렬을 참고한다.

어느것이 옳은가?

14. 다음 명령문은 무엇을 정의하는가?

```
Employee emplist[1000];
```

15. 배렬 emplist의 17번째 원소인 구조체변수에서 salary라는 구조체성원을 호출하는 식을 쓰시오.

16. 탄창에서 처음으로 보관되는 자료항목은

① 첨수를 가지지 않는다.

② 첨수 0을 가진다.

③ 처음으로 삭제되는 자료항목이다.

④ 마감에 삭제되는 자료항목이다.

어느것이 옳은가?

17. Bird형객체 50개를 보관하는 배렬 manyBirds를 정의하는 명령문을 쓰시오.

18. 번역프로그람은 10원소의 배렬의 14번째 원소를 호출하려고 할 때 오유를 통보하는가?

19. 배렬 manyBirds의 27번째 원소인 클라스 Bird의 객체의 성원함수 Cheep()를 실행하는 명령문을 쓰시오.

20. C++의 문자렬은 어떤 형의 배렬인가?

21. 20개까지의 문자를 보관하는 문자렬변수 city를 정의하는 명령문을 쓰시오.

22. 값 "C6H1206-H20"을 가지는 dextrose라는 문자렬상수를 정의하는 명령문을 쓰시오.

23. 발취연산자(>>)는 공백과 만날 때 문자렬읽기를 계속하는가?

24. 다음과 같은 방법으로 여러행의 본문으로 이루어진 입력을 읽어들일수 있다.

① 보통 cout와 <<을 결합하는 방법

② 1인수를 가지는 cin.get()함수

③ 2인수를 가지는 cin.get()함수

④ 3인수를 가지는 cin.get()함수

어느것이 옳은가?

25. 문자렬서고함수를 사용하여 문자렬 name을 문자렬 blank에 복사하는 명령문

을 쓰시오.

26. 두개의 자료성원 bread문자렬과 age라는 int형성원을 가지는 클라스 dog의 선언을 쓰시오.

27. 프로그람을 쓸 때 표준 C++와 C문자렬중 어느것을 사용하여야 하는가?

28. string클라스의 객체들은

① 0으로 끝난다.

② 대입연산자로 복사할수 있다.

③ 기억관리를 요구하지 않는다.

④ 성원함수를 가지지 않는다.

어느것이 옳은가?

29. 문자렬 s1의 어느 위치에 문자렬 "cat"가 있는가를 찾는 명령문을 쓰시오.

30. 문자렬 "cat"를 문자렬 s1의 12위치에 삽입하는 명령문을 쓰시오.

련습문제

1. C문자렬을 반전하는 Reversit()함수를 쓰시오. for순환을 사용하여 첫 문자와 마지막문자를 바꾸고 둘째와 마지막 둘째문자를 바꾸는 방법으로 조작하시오. 문자렬은 Reversit()에 인수로 넘겨야 한다. Reversit()를 시험하기 위한 프로그람을 쓰시오. 프로그람은 사용자로부터 문자렬을 얻어서 Reversit()를 호출하고 결과를 출력한다. 문자렬안에 공백을 포함하는 입력방법을 사용하시오. "Computer science"를 가지고 시험하시오.

2. string클라스의 객체인 이름과 long형의 종업원번호를 포함하는 Employee 클라스를 창조하시오. 객체에 삽입하기 위한 자료를 사용자로부터 얻는 GetData()라는 성원함수, 자료를 표시하는 PutData()라는 성원함수를 정의하시오. 문자렬안에 공백이 없는 이름을 사용하시오. 이 클라스를 시험하는 프로그람을 쓰시오. Employee형배렬을 창조하고 100명의 종업원자료를 사용자가 입력하게 하시오. 끝으로 모든 종업원자료를 출력하시오.

3. 사용자가 입력한 100개의 거리값의 평균을 계산하는 프로그람을 쓰시오. 실례 7-9에서와 같이 Distance클라스의 객체배렬을 창조하시오. 평균을 계산하기 위하여 실례 6-7에서 AddDist()성원함수를 사용할수 있다. 또한 Distance값을 옹근수로 나누는 성원함수들을 요구한다. 여기에 함수가 하나 있다.

```
void Distance::DivDist(Distance d2, int divisor)
{
   float fltCench = d2.cenchies + d2.meters / 100.0;
   fltCench /= divisor;
   cenchies = int(fltCench);
```

```
    meters = (fltCench - cenchies) * 100.0;
  }
```

4. 옹근수를 입력하여 그것들을 int배렬에 보관하는 프로그람을 쓰시오. 배렬을 한 원소씩 횡단하면서 최대값을 찾는 MaxInt()를 쓰시오. 이 함수는 배렬의 주소와 원소개수를 인수로 가지며 최대원소의 첨수를 돌려준다. 프로그람은 이 함수를 호출하고 최대원소와 그 첨수를 표시한다. (실례 7-4～7-6)

5. 6장의 련습 11, 12로부터 Fraction클라스를 쓰시오. 사용자로부터 분수를 얻어서 Fraction형의 배렬에 보관하고 그것들을 평균한 다음 결과를 표시하는 프로그람을 쓰시오.

6. 카드를 사용하여 4명이 오락을 하는 경기에서 매 선수는 13개의 카드를 가진다. 실례 7-10을 변경하여 한조의 카드를 치게 한 다음 4명이 카드를 나누어 들었을 때 4선수의 매개 손에 들고있는 카드들을 표시하는 프로그람을 쓰시오.

7. 사무처리프로그람을 쓰는데서 C++의 약점의 하나는 173,698,001.32원과 같은 화페량의 내부형을 포함하지 않는것이다. 그러한 화페형은 고정소수점수와 17자리의 정확도로 보관하며 반점과 W으로 금액을 충분히 관리할수 있어야 한다. C++기본형 long double형이 19자리의 정확도를 가지므로 화페클라스의 기초로서 그것을 사용할수 있다. 그러나 화페량을 입출력한 후에 화페문자가 있어야 하며 세자리씩 반점으로 구분해야 한다. 이것은 큰 량의 수를 쉽게 읽을수 있게 한다.

클라스를 쓸 때 우선 1,224,567,890,123.99원과 같은 금액을 표시하는 화페문자렬을 인수로 가지는 MsToLd()함수를 쓰시오. 이 함수는 long double값을 보낸다. 화페문자렬은 문자들의 배렬로서 다룰수 있다. 그리고 한문자씩 1～9까지의 수자와 소수점만 다른 문자렬에 복사한다. 화페문자와 반점을 무시하시오. 그다음 _atold()서고함수를 사용하여 결과문자렬을 long double형으로 변환하시오. 화페값은 부수일수 없다. 사용자로부터 화페문자렬을 반복적으로 얻어서 MsToLd()를 시험하고 long double값을 표시하는 main()프로그람을 쓰시오.

8. C++는 배렬첨수가 한계에 이르렀는가를 검사하지 못한다. 이것은 배렬조작을 빠르게 하지만 안전성을 떨군다. 모든 배렬호출의 첨수를 검사하는 안전한 배렬을 창조하는 클라스를 정의하시오. 유일한 자료성원으로서 고정된 크기(LIMIT)의 int배렬을 사용하는 SafeArray라는 클라스를 쓰시오. 성원함수는 두개이다. 우선 PutEl()은 첨수와 int값을 인수로 가지며 그 첨수의 배렬에 int값을 삽입한다. 둘째 성원함수 GetEl()은 인수로서 첨수를 가지며 그 첨수를 가지는 원소의 int값을 돌려준다.

```
SafeArray sa1;           // SafeArray객체를 창조한다.
int temp = 12345;        // int값을 정의한다.
sa1.PutEl(7, temp);      // temp값을 첨수 7의 배렬에 삽입한다.
temp = sa1.GetEl(7);     // 첨수 7의 값을 배렬로부터 얻는다.
```

두 함수는 첨수인수가 0보다 작거나 LIMIT-1보다 큰가를 확인하여야 한다. 기억

기의 다른 부분에 쓰지 않고 이 배렬을 사용할수 있다.

9. 대기렬은 탄창과 매우 류사한 자료보관용기이다. 차이는 탄창에서는 마지막으로 보관된 자료가 처음 얻어지지만 대기렬에서는 첫째 자료항목이 처음에 얻어지는것이다. 즉 탄창은 LIFO수법을 사용하지만 대기렬은 FIFO수법을 사용한다. 대기렬은 은행에서 손님들의 줄과 같다. 대기렬에 먼저 들어온 손님에게 먼저 봉사한다. 실례 7-8에서 Stack라는 클라스대신에 Queue라는 클라스에 대응하도록 변경을 가하시오. 구성자와 함께 두개의 함수 즉 대기렬에 자료항목을 넣는 Put()와 대기렬로부터 자료를 얻는 Get()를 가져야 한다. 이것은 Stack클라스의 Push() 및 Pop()와 등가하다. 대기렬과 탄창은 자료를 보관하는데 사용한다. 탄창은 top라는 int형의 단일변수를 가지지만 대기렬에서는 대기렬의 선두를 가리키는 head와 꼬리를 가리키는 tail이라는 변수를 가진다. 항목은 대기렬의 꼬리에 보관하고 대기렬의 머리부에 있는 항목은 삭제한다. 꼬리나 머리가 배렬의 끝에 이르면 선두로 돌아와야 한다. 그러므로 꼬리를 절환하기 위하여 다음의 명령문

```
if(tail == MAX - 1)
   tail = -1;
```

이 요구되며 머리인 경우도 같다. 대기렬에서 사용된 배렬은 머리와 꼬리가 그것을 순환하므로 원형완충기(circular buffer)라고 한다.

10. 행렬은 2차원배렬이다. 련습 8의 배렬클라스와 같은 안전성을 제공하는 Matrix클라스를 창조하시오. 즉 배렬첨수가 한계를 벗어나는가를 확인해야 한다. Matrix클라스는 10×10배렬로 만드시오. 구성자는 프로그람작성자가 행렬의 실제차원(10×10이하)을 지적해야 한다. 행렬의 자료를 호출하는 성원함수들은 두개의 첨수 즉 배렬의 각 차원을 요구한다. 아래에 이 클라스에 대하여 조작하는 실례가 있다. 즉

```
Matrix m1(3,4);           // Matrix객체의 정의
int temp = 12345;         // int 값의 정의
m1.PutEl(7,4,temp);       // temp를 Matrix객체의 7,4위치에 삽입
temp = m1.GetEl(7,4);     // Matrix객체의 7,4위치의 값을 얻기
```

11. 련습 7을 참고하시오. long double형으로 표시된 수를 화페문자렬값으로 역변환하는 함수 LdToMs()를 정의하시오. 먼저 원래의 수가 long double값보다 크지 않는가를 검사해야 한다. 9,999,999,999,999,990.00이상의 수를 변환하지 않아야 한다. 그다음 long double을 기억기에 보관하고 순수문자렬(W 혹은 반점이 없다.)로 변환하시오. 이때 이 장에서 론의한 ostrstream객체를 사용하시오. 그때 생기는 형식화된 문자렬은 ustring이라는 완충기에 보관하시오. 그다음 W기호를 가진 문자렬을 써야 한다. 즉 수자마다 반점을 삽입한다. 또한 빈자리수만큼 0을 제거해야 한다. 실례로 W3,124.95를 표시할 때 W0,000,000,000,003,124.95라고 하지 말아야 한다.

12. BMoney클라스를 창조하시오. BMoney는 화페량을 long double형으로 보관해야 한다. 함수 MsToLd()에 의하여 입력한 화페문자렬을 long double형으로 보관하고

함수 LdToMs()에 의하여 long double을 화폐문자렬로 변환해야 한다. (련습 6, 10) 입출력을 위한 성원함수 GetMoney()와 PutMoney()를 사용할수 있다. 두개의 BMoney 객체를 더하는 성원함수 MAdd()를 정의하시오. BMoney객체들의 더하기는 간단하다. 즉 두 BMoney객체들의 long double성원자료들을 더하시오. 사용자가 두개의 화폐문자렬을 련이어 입력하면 그 합을 화폐문자렬로 표시하는 main()을 쓰시오. 클라스지정자는 다음과 같다.

```
class BMoney
{
private:
   long double money;
public:
   BMoney();
   BMoney(char s[]);
   void MAdd(BMoney m1, BMoney m2);
   void GetMoney();
   void PutMoney();
};
```

제 8 장. 연산자의 재정의

연산자재정의는 객체지향프로그람작성법의 가장 우월한 특성의 하나이다.

연산자재정의는 복잡하고 모호한 프로그람을 직관적이고 명백한것으로 변환한다. 실례로 명령문

```
d3.AddObjects(d1,d2);
```

또는

```
d3 = d1.AddObjects(d2);
```

을 읽기 쉬운

```
d3 = d1 + d2;
```

로 변환할수 있다.

연산자재정의(overloading operator)라는 용어는 표준 C++연산자 +, *, <=, +=들을 사용자정의자료형에 적용할수 있게 추가적인 의미를 부여한다는것을 의미한다. 보통

```
a = b + c;
```

는 int나 float와 같은 기본형에 대해서는 동작하지만 사용자정의클라스의 객체인 a, b, c에 그것을 적용할 때 번역프로그람으로부터 오유를 일으킨다. 그러나 재정의를 사용하면 a, b, c가 사용자정의형일 때에도 옳은 명령문으로 된다.

연산자재정의는 C++언어를 다시 정의할 기회를 준다. C++연산자의 동작에서 제한이 발견되면 그것을 요구대로 변경할수 있다. 클라스를 사용하여 새로운 종류의 변수들을 창조하고 연산자재정의에 의하여 새로운 연산자를 정의함으로써 C++를 확장하고 자기의 새로운 언어를 만들수 있다.

자료형변환이라는 또 하나의 연산은 연산자재정의와 밀접하게 련관되여있다. C++는 int나 float와 같은 기본형의 변환을 자동적으로 조종하지만 사용자정의형을 포함하는 변환은 프로그람작성자에게 일정한 작업을 요구한다.

이 장에서는 단항연산자와 2항연산자의 재정의, 변환구성자의 재정의, 기본형과 사용자정의형사이의 변환, 재정의에서 몇가지 주의할 점에 대하여 설명한다.

제 1 절. 단항연산자의 재정의

단항연산자의 재정의부터 고찰하자. 단항연산자(unary operator)는 오직 하나의 연산수에 대하여 조작한다.

연산수(operand)란 연산자가 조작하는 단순한 변수이다. 단항연산자의 실례로서 증가감소연산자 ++와 --, 단항미누스(실례로 -33) 등이 있다.

실례 6-5에서는 계수과정을 보존하기 위하여 클라스 Counter를 작성하였다.

Counter클라스의 객체들은 성원함수호출에 의하여 증가된다.

```
c1.IncCount();
```

이 명령문대신에 아래와 같이 증가연산자 ++를 사용하면 더 편리하다.

```
++c1;
```

이것을 실현한 실례 8-1을 보기로 하자.

(실례 8-1) ++연산자를 사용하여 계수기변수의 증가

```
#include <iostream>
using namespace std;
class Counter
{
private:
   unsigned int count;
public:
   Counter() : count(0) {}
   unsigned int GetCount() { return count; }
   void operator++ () { ++count; }
};
int main()
{
   Counter c1, c2;
   cout << "\nc1=" << c1.GetCount();
   cout << "\nc2=" << c2.GetCount();
   ++c1;
   ++c2;
   ++c2;
   cout << "\nc1=" << c1.GetCount();
   cout << "\nc2=" << c2.GetCount() << endl;
   return 0;
}
```

이 프로그람에서는 두개의 Counter클라스객체 c1와 c2을 창조한다. 객체안의 count가 0으로 초기화되고 곧 표시된다. 그다음 재정의된 ++연산자에 의하여 c1을 한번, c2을 두번 증가시키고 결과값들을 표시한다. 여기에 프로그람의 출력이 있다.

```
c1=0
c2=0
c1=1
c2=2
```

이 연산에 대응하는 명령문은 다음과 같다.

```
++c1;
++c2;
++c2;
```

++연산자를 c1에 한번, c2에 두번 적용한다. 실례에서는 앞붙이표기를 사용한다.

1. operator예약어

그러면 표준 C++연산자가 사용자정의연산수에 대하여 어떻게 동작하는가?

예약어 operator는 다음의 형식으로 ++연산자를 재정의하는데 사용된다.

```
void operator++();
```

먼저 돌림값형(이 경우에 void), 그 뒤에 예약어 operator, 그 뒤에 연산자(++), 끝으로 인수목록이 괄호안에 놓인다. 이 선언은 번역프로그람에게 연산수가 Counter형인 ++연산자를 만나면 이 성원함수를 호출하게 한다.

5장에서 번역프로그람이 재정의된 함수를 구별하는 유일한 방법은 자료형과 인수개수라는것을 보았다. 마찬가지로 재정의된 연산자를 구별하는 유일한 방법은 그 연산수들의 자료형이다. 연산수가 int와 같은 기본형이면 즉

```
++intVar;
```

번역프로그람은 내부루틴을 사용하여 int를 증가시킨다. 연산수가 Counter형변수이면 번역프로그람은 그대신 사용자가 정의한 operator++()를 사용한다.

2. operator인수

main()에서 ++연산자는 ++c1과 같이 특정한 식에 적용된다. operator++()에는 인수가 없다. ++연산자는 그것이 성원인 객체의 count자료를 증가시킨다. 성원함수는 항상 그것이 성원인 특정한 객체를 호출할수 있으므로 ++연산자는 인수를 요구하지 않는다. 이것을 그림 8-1에서 보여준다.

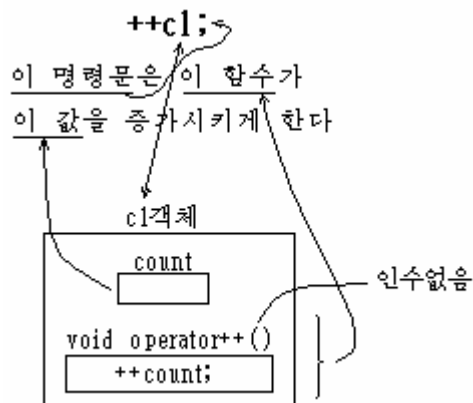



그림 8-1. 인수없는 재정의된 단항연산자

3. operator돌림값

실례 8-1에서 operator++()함수는 미묘한 결함을 가지고있다. main()에서 다음과 같이

```
c1 = ++c2;
```

을 사용하면 그것을 발견할수 있다.

번역프로그람은 오유를 통보한다. 왜냐하면 operator++()함수에서 void형의 돌림값을 가지도록 ++연산자를 정의함으로써 대입명령문에서 Counter형의 변수를 돌려보낼것을 요구하기때문이다. 즉 번역프로그람은 c2이 ++연산자에 대하여 조작한 다음

그것을 돌려주고 값을 c1에 대입할것을 요구한다. 실례 8-1에서는 대입식에서 Counter객체를 증가시키는데 ++를 사용할수 없으므로 항상 연산수와 함께 독자적으로 사용한다. 물론 표준++연산자를 int와 같은 기본형에 적용할수 있다.

대입식에서 자체로 만든 operator++()를 사용하도록 하자면 돌림값을 돌려주어야 한다. 다음의 실례 8-2가 그것을 가능하게 한다.

(실례 8-2) 돌림값을 가지는 ++연산자에 의하여 계수기변수의 증가

```
#include <iostream>
using namespace std;
class Counter
{
private:
  unsigned int count;
public:
  Counter() : count(0) {}
  unsigned int GetCount() { return count; }
  Counter operator++ ()
  {
     ++count;
     Counter temp;
     temp.count = count;
     return temp;
  }
};
int main()
{
  Counter c1, c2;
  cout << "\nc1=" << c1.GetCount();
  cout << "\nc2=" << c2.GetCount();
  ++c1;
  c2 = ++c1;
  cout << "\nc1=" << c1.GetCount();
  cout << "\nc2=" << c2.GetCount() << endl;
  return 0;
}
```

여기서 operator++()함수는 Counter형의 새로운 객체 temp를 창조하고 돌림값으로 사용한다. 이 함수는 사전에 자기 객체의 count자료를 증가시키고 새로운 temp객체를 창조하며 그 count에 자기 객체의 값과 같은 값을 대입한다.

끝으로 temp객체를 돌려준다. 이것은 요구하는 효과를 가진다. 식

```
++c1;
```

은 값을 돌려보내므로 다른 식에서 사용할수 있다. 즉 main()에서와 같이 명령문

```
c2 = ++c1;
```

에서는 c1++가 돌려준 값을 c2에 대입한다. 이 프로그람의 출력은 다음과 같다.

```
c1=0
```

```
c2=0
c1=2
c2=2
```

4. 이름없는 림시객체

실례 8-2에서는 temp라는 Counter형의 림시객체를 창조한다. 목적은 ++연산자용의 돌림값을 제공하는데 있다. 이것은 세개의 명령문을 요구한다.

```
Counter temp;
temp.count = count;
return temp;
```

함수와 재정의된 연산자로부터 림시객체를 돌려주는 일반적인 방법이 있다. 실례 8-3에서 보여준 수법을 시험하자.

(실례 8-3) 이름없는 림시객체를 사용하는 ++연산자에 의한 계수기변수의 증가

```
#include <iostream>
using namespace std;
class Counter
{
private:
  unsigned int count;
public:
  Counter() : count(0) {}
  Counter(int c) : count(c) {}
  unsigned int GetCount() { return count; }
  Counter operator++ ()
  {
     ++count;
     return Counter(count);
  }
};
int main()
{
  Counter c1, c2;
  cout << "\nc1=" << c1.GetCount();
  cout << "\nc2=" << c2.GetCount();
  ++c1;
  c2 = ++c1;
  cout << "\nc1=" << c1.GetCount();
  cout << "\nc2=" << c2.GetCount() << endl;
  return 0;
}
```

이 프로그람에서는 단일명령문

```
return Counter(count);
```

이 실례 8-2의 세개의 명령문 모두가 수행하던 동작을 수행한다. 이 명령문은 이름없는 Counter형의 객체를 창조한다. 즉 아주 간단하다. 이름없는 객체는 인수 count가 제공하는 값으로 초기화된다.

그런데 이것은 1인수구성자를 요구한다. 이 명령문이 동작하게 하기 위하여 실례 8-3에서 1인수구성자를 성원함수로서 추가한다.

```
Counter(int c) : count(c) {}
```

이름없는 객체가 count의 값으로 초기화되면 그것을 돌려줄수 있다. 프로그람의 출력은 실례 8-2와 같다.

실례 8-2와 실례 8-3의 수법들은 원시객체(함수가 성원인 객체)의 사본을 만들고 돌려주게 한다.

5. 뒤붙이표기

지금까지는 앞붙이형식의 증가연산자 즉 ++c1을 보았다.

그러면 변수값을 사용한 다음 증가하는 뒤붙이형식 즉 c1++에 대하여 고찰하자

두가지 형식의 증가연산자가 동작하게 하려면 실례 8-4에서 보여주는것처럼 두개의 재정의된 ++연산자를 정의해야 한다.

(실례 8-4) 앞붙이와 뒤붙이형식의 재정의된 ++연산자

```
#include <iostream>
using namespace std;
class Counter
{
private:
   unsigned int count;
public:
   Counter() : count(0) {}
   Counter(int c) : count(c) {}
   unsigned int GetCount() const { return count; }
   Counter operator++ () // 앞붙이
   {
      return Counter(++count);
   }
   Counter operator++ (int)   // 뒤붙이
   {
      return Counter(count++);
   }
};
int main()
{
   Counter c1, c2;
   cout << "\nc1=" << c1.GetCount();
   cout << "\nc2=" << c2.GetCount();
   ++c1;       // c1=1
```

```
    c2 = ++c1;// c1=2,c2=2 (앞붙이)
    cout << "\nc1=" << c1.GetCount();
    cout << "\nc2=" << c2.GetCount();
    c2 = c1++;// c1=3,c2=2 (뒤붙이)
    cout << "\nc1=" << c1.GetCount();
    cout << "\nc2=" << c2.GetCount() << endl;
    return 0;
  }
```

++연산자를 재정의하는 두개의 서로 다른 선언자들이 있다. 그 하나는 앞붙이형식이다. 즉

```
Counter operator++ ()    // 앞붙이
```

또 하나는 뒤붙이형식이다. 즉

```
Counter operator++ (int)// 뒤붙이
```

유일한 차이는 괄호안의 int이다. 여기서 int는 실제로 인수가 아니고 옹근수를 의미하지도 않는다. 그것은 단순히 뒤붙이형식의 연산자를 창조한다는것을 번역프로그람에게 알린다. 여기에 프로그람의 출력이 있다.

```
c1=0
c2=0
c1=2
c2=2
c1=3
c2=2
```

실례 8-2와 실례 8-3의 처음 4행의 출력을 보았다.

그러나 마지막 2행에서는 c2 = c1++명령문의 결과를 보았다.

여기서 c1은 3으로 증가하지만 c2은 그것이 증가하기 전에 c1의 값에 대입되므로 값 2를 가진다.

물론 감소연산자에 대해서도 이와 같은 수법을 적용할수 있다.

제 2 절. 2항연산자의 재정의

2항연산자(binary operator)는 단항연산자처럼 간단히 재정의할수 있다. 그러면 산수연산자, 비교연산자, 산수대입연산자의 재정의를 고찰하자.

1. 산수연산자

실례 6-7에서 두개의 Distance객체들을 성원함수 AddDist()에 의하여 더하는 방법을 보았다.

```
dist3.AddDist(dist1, dist2);
```

+연산자를 재정의하면 이 식을 다음과 같이 간략할수 있다.

```
dist3 = dist1 + dist2;
```

이것을 리용한것이 실례 8-5이다.

(실례 8-5) 두개의 Distance를 더하는 +연산자의 재정의

```
#include <iostream>
using namespace std;
class Distance
{
private:
  int meters; float centies;
public:
  Distance() : meters(0), centies(0.0) {}
  Distance(int mm, float cc) : meters(mm), centies(cc) {}
  void GetDist()
  {
     cout << "\n메터를 입력하십시오:"; cin >> meters;
     cout << "\n센치메터를 입력하십시오:"; cin >> centies;
  }
  void ShowDist() const { cout << meters << "m " << centies << "㎝"; }
  Distance operator+ (Distance) const;
};
Distance Distance::operator+ (Distance d2) const
{
  int m = meters + d2.meters;
  float c = centies + d2.centies;
  if(c >= 100.0) { c -= 100.0; m++; }
  return Distance(m, c);
}
int main()
{
  Distance dist1, dist3, dist4;
  dist1.GetDist();
  Distance dist2(11, 6.25);
  dist3 = dist1 + dist2;
  dist4 = dist1 + dist2 + dist3;
  cout << "dist1="; dist1.ShowDist(); cout << endl;
  cout << "dist2="; dist2.ShowDist(); cout << endl;
  cout << "dist3="; dist3.ShowDist(); cout << endl;
  cout << "dist4="; dist4.ShowDist(); cout << endl;
  return 0;
}
```

더하기결과를 대입에서처럼 다중더하기에서 사용할수 있다는것을 보여주기 위하여 또 다른 더하기를 main()에서 처리한다. dist1, dist2, dist3을 더하여 dist4를 얻자면 즉

```
dist4 = dist1 + dist2 + dist3;
```

여기에 프로그람의 출력이 있다.

```
메터를 입력하십시오:10
센치메터를 입력하십시오:6.5
dist1=10m 6.5cm
```

dist2=11m 6.25cm
dist3=21m 12.75cm
dist4=42m 25.5cm

클라스 Distance에서 operator+()함수선언은 다음과 같다.

```
Distance operator+ (Distance) const;
```

이 함수는 Distance형의 돌림값과 Distance형의 한개 인수를 가진다.

```
dist3 = dist1 + dist2;
```

와 같은 식에서 연산자의 돌림값과 인수들이 객체들과 어떻게 련결되는가를 리해하여야 한다. 번역프로그람이 이런 식과 만나면 인수형을 검사하여 Distance형을 발견하고 Distance성원함수 operator+()를 실행한다.

그러면 이 함수의 인수는 무엇인가? dist1인가 dist2인가?

operator+()는 두개의 수를 더하는데 두개의 인수를 요구하지 않는다.

여기에 열쇠가 있다. 연산자의 왼변에 있는 객체(이 경우에 dist1)은 연산자가 성원인 객체이다. 연산자의 오른변에 있는 객체(dist2)은 연산자에 인수로서 공급된다. 연산자는 값을 돌려주고 그것은 다음 연산에 사용된다. 즉 이 경우에 dist3에 대입된다. 그림 8-2에 그것을 보여준다.

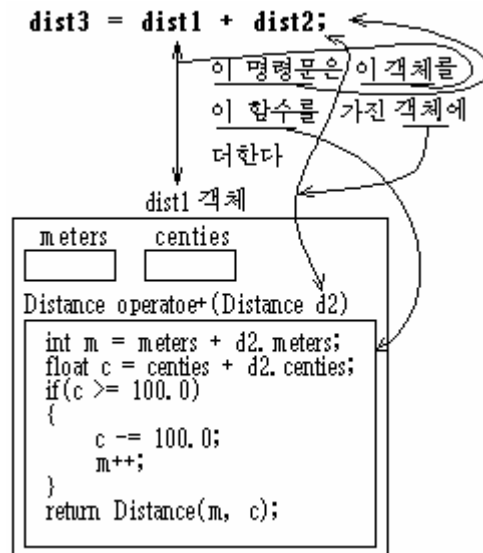



그림 8-2. 재정의된 2항연산자(1인수)

operator+()함수에서 왼쪽 연산수는 meters와 centies에 의하여 직접 호출된다. 왼쪽 연산수는 연산수가 성원인 객체이다. 오른쪽 연산수는 함수의 인수로서 즉 d2.meters와 d2.centies에 의하여 호출된다.

일반적으로 말하면 한개 연산수는 연산자가 성원인 객체이므로 재정의된 연산자는 연산수의 수보다 하나 적은 인수를 요구한다. 이 규칙은 동료함수와 연산자에 대해서는 적용할수 없다.

실례 8-5에서 operator+()의 돌림값을 계산하기 위하여 두 연산수로부터 meters와 centies를 먼저 더하고 그 결과값 m과 c에 의하여 이름없는 Distance객체를 초기화하고 그것을 명령문

```
return Distance(m, c);
```

에 의하여 돌려준다.

이것은 실례 8-3에서 사용한 배렬과 비슷하다. 다만 구성자가 한개 인수대신 두개 인수를 가진다는것만 다르다. 그다음 main()에서 명령문

```
dist3 = dist1 + dist2;
```

은 이름없는 Distance객체의 값을 dist3에 대입한다. 이 명령문을 실례 6-7에서와 같은 과제를 처리하기 위한 함수호출과 비교할수 있다.

류사한 방법으로 -, *, /와 같은 다른 연산자들을 재정의할수 있다.

2. 문자렬련결

+연산자를 C문자렬련결에 사용할수 없다. 즉

```
str3 = str1 + str2;
```

라고 할수 없다. 여기서 str1, str2, str3은 "cat" + "bird" = "catbird"에서와 같이 C문자렬변수(char형배렬)이다. 그러나 실례 7-20에서처럼 자체의 String클라스를 사용한다면 +연산자를 재정의하여 그러한 련결을 처리할수 있다. 이것은 표준 C++ string클라스를 통하여 작업과정을 고찰하는것이 더 리해하기 쉽다. 엄격히 더하기가 아닌 어떤 일을 수행하도록 +연산자를 재정의하는것이 바로 C++에서 재정의의 실례이다. 여기에 실례 8-6이 있다.

(실례 8-6) 문자렬을 련결하는 +연산자의 재정의

```
#include <iostream>
using namespace std;
#include <string.h>
#include <stdlib.h>
class String
{
private:
   enum { SZ = 80 };
   char str[SZ];
public:
   String() { strcpy(str, ""); }
   String(char s[]) { strcpy(str, s); }
   void Display() const { cout << str; }
   String operator+ (String ss) const
   {
      String temp;
      if(strlen(str) + strlen(ss.str) < SZ)
      {
         strcpy(temp.str, str);
```

```
            strcat(temp.str, ss.str);
        }
        else
        {
            cout << "\n문자렬 넘침";
            exit(1);
        }
        return temp;
    }
};
int main()
{
    String s1 = "\n동무들! ";
    String s2 = "새해를 축하합니다!";
    String s3;
    s1.Display(); s2.Display(); s3.Display();
    s3 = s1 + s2;
    s3.Display(); cout << endl;
    return 0;
}
```

프로그람은 우선 세개의 문자렬을 하나씩 표시한다. 세번째는 현재 비여있으므로 아무것도 출력되지 않는다. 그다음 처음 두개의 문자렬을 련결하여 세번째 문자렬에 보관한다. 그리고 세번째 문자렬을 다시 표시한다. 여기에 그 출력이 있다.

```
동무들! 새해를 축하합니다!          ← s1,s2,s3(빔)
동무들! 새해를 축하합니다!          ← 련결후 s3
```

+연산자재정의의 기초는 비슷하다. 선언자

```
String operator+ (String ss) const
```

에서는 +연산자가 String형의 한개 인수를 가지며 같은 형의 객체를 돌려준다. operator+()에서는 String형의 림시객체를 창조하고 자체의 String객체의 문자렬을 거기에 복사하며 서고함수 strcat()를 사용하여 인수문자렬을 련결하고 결과로 얻어지는 림시문자렬을 돌려준다.

다음의 명령문

```
return String(string);
```

을 사용할수 없다. 여기서는 이름없는 림시 String객체가 창조되므로 그것을 초기화할뿐이다. 인수문자렬과 련결하려면 String형의 림시객체를 호출해야 한다.

String클라스에서는 고정길이문자렬이 넘어나지 않도록 주의하여야 한다. operator+()함수에서 그러한 경우를 방지하기 위하여 두개 문자렬의 련결길이가 최대문자렬길이를 초과하지 않는가를 검사하고 초과하면 련결조작을 수행하지 않고 오유통보를 출력한다. 물론 다른 방법으로 오유를 처리할수도 있다.

enum을 사용하여 상수값 SZ를 정의한다. 모든 번역프로그람이 표준 C++에 준할때 다음과 같이 변경할수 있다.

```
static const int SZ = 80;
```

+연산자의 여러가지 사용방법 즉 거리의 더하기와 문자렬련결을 고찰하였다. 이 클라스들을 모두 하나의 프로그람에 넣는 경우에도 C++는 오유없이 처리한다.

번역프로그람은 정확한 함수를 선택하고 연산수의 형에 기초하여 더하기를 수행한다.

3. 비교연산자

다른 종류의 C++연산자 즉 비교연산자의 재정의방법을 고찰하자.

1) 거리비교

첫 실례에서는 Distance클라스의 작기연산자 <를 재정의하여 두개의 거리를 비교한다.

(실례 8-7) 두개의 Distance를 비교하는 <연산자의 재정의

```
#include <iostream>
using namespace std;
class Distance
{
private:
   int meters;
   float centies;
public:
   Distance() : meters(0), centies(0.0) {}
   Distance(int mm, float cc) : meters(mm), centies(cc) {}
   void GetDist()
   {
      cout << "\n메터를 입력하십시오:"; cin >> meters;
      cout << "\n센치메터를 입력하십시오:"; cin >> centies;
   }
   void ShowDist() const { cout << meters << "m " << centies << "㎝"; }
   bool operator< (Distance) const;
};
bool Distance::operator< (Distance d2) const
{
   float bf1 = meters + centies / 100;
   float bf2 = d2.meters + d2.centies / 100;
   return (bf1 < bf2) ? true : false;
}
int main()
{
   Distance dist1;
   dist1.GetDist();
   Distance dist2(11, 6.25);
   cout << "dist1="; dist1.ShowDist(); cout << endl;
   cout << "dist2="; dist2.ShowDist(); cout << endl;
   if(dist1 < dist2)
      cout << "\ndist1 < dist2";
```

```
    else
        cout << "\ndist1 >= dist2";
    cout << endl;
    return 0;
}
```

이 프로그람은 사용자가 입력한 거리를 프로그람에서 초기화한 거리 6m 2.5cm와 비교한다. 결과에 따라 두개의 가능한 문장중 하나를 출력한다. 여기에 대표적인 출력이 있다.

```
메터를 입력하십시오:5
센치메터를 입력하십시오:11.5
dist1=5m 11.5cm
dist2=11m 6.25cm
dist1 < dist2
```

실례 8-7에서 operator<()함수에서 사용한 수법은 operator<()함수가 bool형의 돌림값을 가진다는것을 제외하면 실례 8-5에서 +연산자재정의와 비슷하다. 돌림값은 두 거리의 비교에 따라서 false 혹은 true이다. 이 연산은 두 거리를 류동소수점수형의 센치메터로 변환하고 그것들을 표준 <연산자를 사용하여 비교함으로써 이루어진다.

조건연산자의 사용

```
return (bf1 < bf2) ? true : false;
```

은

```
if (bf1 < bf2)
    return true;
else
    return false;
```

와 같다.

2) 문자렬비교

연산자재정의의 다른 실례가 있다. 이번에는 같기(==)연산자이다. 자체로 만든 두개의 String객체를 비교하여 같으면 true, 같지 않으면 false를 돌려준다.

(실례 8-8) 문자렬을 비교하는 ==연산자의 재정의

```
#include <iostream>
using namespace std;
class String
{
private:
    enum { SZ = 100 };
    char str[SZ];
public:
    String() { strcpy(str, ""); }
    String(char s[]) { strcpy(str, s); }
    void Display() const { cout << str; }
    void GetStr() { cin.get(str, SZ); }
    bool operator== (String ss) const
      { return (strcmp(str, ss.str) == 0) ? true : false; }
```

```
};
int main()
{
   String s1 = "yes";
   String s2 = "no";
   String s3;
   cout << "\n'yes' 혹은 'no'라고 입력하십시오:";
   s3.GetStr();
   if(s3 == s1)
       cout << "'yes'라고 입력하였습니다.\n";
   else  if(s3 == s2)
       cout << "'no'라고 입력하였습니다.\n";
  else
       cout << "지령대로 입력하지 않았습니다.\n";
   return 0;
}
```

이 프로그람의 main()부분에서는 ==연산자를 두번 사용한다. 한번은 사용자에 의한 문자렬입력이 "yes"인가를 알아보기 위하여, 또 한번은 그것이 "no"인가를 알아보는데 사용한다. 여기에 사용자가 "yes"라고 입력할 때의 출력이 있다.

```
'yes' 혹은 'no'라고 입력하십시오:yes
'yes'라고 입력하였습니다.
```

operator==()함수는 서고함수 strcmp()에 의하여 두개의 C문자렬을 비교한다. 이 함수는 문자렬이 같으면 0, 첫 문자렬이 둘째 문자렬보다 작으면 부수, 첫째가 둘째보다 크면 정수를 보내온다. 여기서 《작다, 크다》라는것은 첫 문자렬이 자모순으로 둘째 문자렬의 앞에 혹은 뒤에 나타나는가를 가리키는데 사용된다. <와 >와 같은 다른 비교연산자들도 문자렬의 자모순크기를 비교하는데 사용할수 있다.

또한 ==비교연산자는 문자렬길이를 비교하도록 재정의할수도 있다. 연산자의 사용방법을 정의하므로 작성자가 정황에 맞게 정의하고 사용해야 한다.

4. 산수대입연산자

산수대입연산자 +=의 재정의를 고찰하자. +=연산자는 대입과 더하기를 하나로 결합한다. 첫째 거리와 둘째 거리를 더하여 결과를 첫째 거리에 넘기는데 +=연산자를 사용한다. 이것은 처음에 보여준 실례 8-5와 비슷하지만 좀 차이가 있다. 실례 8-9의 코드가 있다.

(실례 8-9) +=대입연산자의 재정의

```
#include <iostream>
using namespace std;
class Distance
{
private:
   int meters;
   float centies;
```

```
public:
   Distance() : meters(0), centies(0) { }
   Distance(int me, float ce) : meters(me), centies(ce) { }
   void GetDist()
   {
      cout << "\n메터를 입력하십시오:"; cin >> meters;
      cout << "센치메터를 입력하십시오:"; cin >> centies;
   }
   void ShowDist() { cout << meters << "m " << centies << "cm"; }
   void operator+=(Distance);
};
void Distance::operator+=(Distance d2)
{
   meters += d2.meters;
   centies += d2.centies;
   if(centies >= 100.0)
   {
      centies -= 100.0;
      meters++;
   }
}
int main()
{
   Distance dist1;
   dist1.GetDist();
   cout << "\ndist1="; dist1.ShowDist();
   Distance dist2(11, 6.25);
   cout << "\ndist2="; dist2.ShowDist();
   dist1 += dist2;
   cout << "\n더한후, ";
   cout << "\ndist1="; dist1.ShowDist(); cout << endl;
   return 0;
}
```

실례에서는 사용자로부터 거리를 얻어서 그것을 둘째 거리에 더한다. 둘째 거리는 프로그람에 의하여 11m 6.15cm로 이미 초기화되여있다. 여기에 프로그람과의 대화가 있다.

```
메터를 입력하십시오:3
센치메터를 입력하십시오:5.75
dist1=3m 5.75cm
dist2=11m 6.25cm
더한후, dist1=14m 12cm
```

이 프로그람에서 더하기는 main()안에서

```
dist1 += dist2;
```

에 의해 수행된다. 이것은 dist1과 dist2의 합을 dist1에 보관한다. 여기서 사용된 함수 operator+=()와 실례 8-5에서 사용된 operator+()사이의 차이를 고찰하자. 초기의

operator+()함수에서는 Distance형의 새로운 객체를 창조하고 함수로부터 돌려보내여 3번째 Distance객체에 대입한다. 즉

```
dist3 = dist1 + dist2;
```

실례 8-9의 operator+=()함수에서 합을 보관하는 객체는 함수가 성원인 객체이다. 즉 그것은 객체를 돌려주는데만 사용되는 림시객체가 아니라 값을 주는 meters와 centies이다. operator+=()함수에는 돌림값이 없다. 즉 void형을 돌려준다.

연산자가 식에서 혼자 쓰이고 대입연산자의 결과는 어디에도 대입되지 않으므로 +=연산자는 돌림값을 가질 필요가 없다.

```
dist1 += dist2;
```

복잡한 식에서 이것을 사용하려고 한다면 즉

```
dist3 = dist1 += dist2;
```

돌림값을 주어야 한다. 이것은

```
return Distance(metes,centies);
```

와 같은 명령문으로 operator+=()함수를 끝냄으로써 실현할수 있다. 여기서는 이름없는 객체를 그 객체와 같은 값으로 초기화하여 돌려준다.

5. 첨수연산자

첨수연산자를 배렬원소호출에 사용할수 있도록 재정의할수 있다. 이것은 C++에서 배렬의 작업방법을 변경하는데 사용할수 있다. 실례로 《안전한》 배렬을 만들수 있다. 안전한 배렬은 첨수가 배렬한계를 벗어나지 않게 호출하는가를 자동적으로 검사한다. (vector클라스를 사용할수 있다.)

재정의된 첨수연산자를 보여주기 위하여 5장의 참고에 의한 값돌려주기를 상기하자. 재정의된 첨수연산자는 참고에 의해 귀환하여야 한다. 왜 그런가?

그것을 리해하기 위하여 안전한 배렬을 실현하는 실례를 고찰하자.

배렬원소들을 삽입하고 읽어들이는데 여러가지 수법을 사용할수 있다. 즉

· put()와 get()함수에 의한 분리

· 참고에 의한 귀환을 사용하는 단일한 access()함수

· 참고에 의한 귀환을 사용하는 재정의된 []연산자

다음 세개의 프로그람은 모두 SafeArray라는 클라스를 창조한다. 그 성원자료는 100개의 int값의 배렬이고 세개가 모두 배렬호출이 한계안에 있는가를 검사한다. 이 프로그람에서 main()함수는 안전한 배렬에 값들을 채워넣고 클라스를 시험한 다음 그것들을 모두 표시하여 모든것이 제대로 동작한다는것을 사용자에게 담보한다.

1) get()와 put()함수의 분리

첫 실례는 배렬원소를 호출하는 두개의 함수 즉 배렬에 값을 삽입하는 PutEl()과 배렬원소의 값을 얻는 GetEl()을 정의한다. 두 함수는 첨수가 배렬범위를 벗어나지 않는다는것 즉 첨수가 0보다 작지 않고 배렬크기보다 크지 않도록 값이 제공되는가를

검사한다.

(실례 8-10) 안전한 배렬만들기

```
#include <iostream>
using namespace std;
#include <process.h>
const int LIMIT = 100;
class SafeArray
{
private:
   int arr[LIMIT];
public:
   void PutEl(int n, int elValue)
   {
      if(n < 0 || n >= LIMIT)
      {
          cout << "\n첨수가 한계를 초과합니다";
          exit(1);
      }
      arr[n] = elValue;
   }
   int GetEl(int n) const
   {
      if(n < 0 || n >= LIMIT)
      {
          cout << "\n첨수가 한계를 초과합니다";
          exit(1);
      }
      return arr[n];
   }
};
int main()
{
   SafeArray sa1;
   for(int j=0; j<LIMIT; j++)
      sa1.PutEl(j, j * 10);
   for(j=0; j<LIMIT; j++)
   {
      int temp = sa1.GetEl(j);
      cout << j << "번째 원소는 " << temp << endl;
   }
   return 0;
}
```

PutEl()성원함수에 의해 안전한 배렬에 자료가 삽입되고 GetEl()에 의해 표시된다. 이것은 안전한 배렬을 실현한다. 만일 범위를 벗어나는 첨수를 사용하려고 시도하면 작성자는 오유통보문을 받아들인다. 그러나 형식은 그리 좋지 않다.

2) 참고에 의해 귀환하는 단일한 access()함수

이번에는 안전한 배렬에 자료를 삽입하고 그것을 읽어내는데 하나의 성원함수를 사용한다. SafeArray의 Access()함수는 참고에 의하여 값을 돌려준다. 이것은 갈기식의 왼변에 함수를 놓을수 있고 오른변값을 함수에 의해 돌아오는 변수에 대입할수 있다는것을 의미한다. 여기에 실례 8-11이 있다.

(실례 8-11) 안전한 배렬변경

```
#include <iostream>
using namespace std;
#include <process.h>
const int LIMIT = 100;
class SafeArray
{
private:
  int arr[LIMIT];
public:
  int& Access(int n)
  {
    if(n < 0 || n >= LIMIT)
    {
        cout << "\n첨수가 한계를 초과합니다";
        exit(1);
    }
    return arr[n];
  }
};
int main()
{
  SafeArray sa1;
  for(int j=0; j<LIMIT; j++)
    sa1.Access(j) = j * 10;
  for(j=0; j<LIMIT; j++)
  {
    int temp = sa1.Access(j);
    cout << j << "번째 원소는 " << temp << endl;
  }
  return 0;
}
```

명령문

```
sa1.access(j) = j * 10;// 갈기기호의 왼변
```

은 함수의 돌림값 arr[j]에 값 j*10을 보관한다.

SafeArray의 입력과 출력에 서로 다른 함수를 사용하는것보다 하나의 함수를 사용하는것이 일반적이고 이름을 기억하기가 쉽다. 그러나 이름을 전혀 기억하지 않아도 되는 좋은 방법이 있다.

3) 참고에 의하여 귀환하는 []연산자의 재정의

표준 C++배렬들에서와 같은 첨수연산자를 사용하여 안전한 배렬을 호출하기 위하

여 SafeArray클라스에서 첨수연산자를 재정의한다. 그러나 일반적으로 첨수연산자는 같기기호의 왼변에 많이 사용되므로 재정의된 함수는 참고에 의해 귀환해야 한다.

(실례 8-12) 안전한 배렬만들기

```
#include <iostream>
using namespace std;
#include <process.h>
const int LIMIT = 100;
class SafeArray
{
private:
   int arr[LIMIT];
public:
   int& operator[](int n)
   {
      if(n < 0 || n >= LIMIT)
      {
         cout << "\n첨수가 한계를 초과합니다";
         exit(1);
      }
      return arr[n];
   }
};
int main()
{
   SafeArray sa1;
   for(int j=0; j<LIMIT; j++)
      sa1[j] = j * 10;
   for(j=0; j<LIMIT; j++)
   {
      int temp = sa1[j];
      cout << j << "번째 원소는 " << temp << endl;
   }
   return 0;
}
```

이 프로그람에서는 자연적인 첨수식

```
sa1[j] = j * 10;
```

과 안전한 배렬에 입출력하는데

```
temp = sa1[j];
```

를 사용할수 있다.

제 3 절. 자료변환

이미 =연산자가 어떤 변수로부터 다른 변수에 값을 대입한다는것을 알고있다. 즉

```
intVar1 = intVar2;
```

여기서 intVar1, intVar2는 옹근수이다. 마찬가지로 다음의 명령문

```
dist3 = dist1 + dist2;
```

와 같이 같은 형으로 주어진 어떤 사용자정의객체의 값을 다른 객체의 값에 대입할수 있다는것을 알고있다.

우에서 더한 결과는 Distance형이고 Distance형의 다른 객체 dist3에 대입된다. 보통 어떤 객체의 값을 같은 형의 다른 객체에 대입할 때 모든 성원자료항목들의 값은 새로운 객체에 단순히 복사된다. 번역프로그람은 Distance객체와 같은 사용자정의 객체들의 대입에 =를 사용하는 다른 특수한 지령을 요구하지 않는다.

이와 같이 =기호의 량변에 같은 자료형으로 주어지는 기본형이나 사용자정의형의 자료형들사이의 대입은 번역프로그람에 의해 조종된다.

그러나 =의 량변에 있는 변수들이 서로 다른 형이라면 어떻게 되겠는가?

우선 번역프로그람이 기본형의 변환을 어떻게 조종하고 어떤 변환이 자동적으로 수행되는가를 고찰한다.

그다음 번역프로그람이 자동적으로 조종하지 못하는 경우 작성자가 수행해야 하는 몇가지 상황을 고찰한다. 여기에는 기본형과 사용자자료형사이의 변환과 서로 다른 사용자자료형사이의 변환이 포함된다.

어떤 형을 다른 형으로 루틴적으로 변환하려는것은 빈약한 프로그람작성법이라고 생각할수 있다. Pascal과 같은 언어들에서 그러한 변환을 수행하려면 상당히 애를 먹는다. 그러나 C++와 C는 변환을 가능하게 한다.

1. 기본형사이의 변환

다음의 명령문이 있다.

```
intVar = floatVar;
```

여기서 intVar는 int형, floatVar는 float형이고 번역프로그람이 특수한 루틴을 호출하여 류동소수점수형식으로 표시된 floatVar를 intVar에 대입할수 있도록 옹근수형식으로 변환한다고 가정한다.

물론 이러한 변환은 많다. 즉 float로부터 double로, char로부터 float로,…

이러한 변환은 자체의 루틴을 가지고있고 그 루틴은 번역프로그람에 짜넣어져있으며 =기호의 서로 다른 변에 다른 자료형들이 놓일 때 호출된다. 이러한 변환은 프로그람으로 표시되지 않으므로 암시적변환(implicit conversion)이라고 한다.

때로는 어떤 형을 다른 형으로 변환할것을 번역프로그람에게 강요한다. 이때 강제형변환연산자를 사용한다. 실례로 float를 int로 변환하려면

```
intVar = static_cast<int>(floatVar);
```

강제형변환은 명시적변환(explicit conversion)을 제공한다. float를 int로 변환하도록 하기 위하여 프로그람에 명백히 static_cast<int>()라고 쓴다. 그렇지만 명시적변환은 암시적변환과 같은 내부루틴을 사용한다.

2. 객체와 기본형사이의 변환

사용자정의자료형과 기본형사이의 변환을 할 때 내부변환루틴에 기초할수 없다. 그것은 번역프로그람의 사용자정의형에 대하여 변환방법을 모르기때문이다.

그대신 자체로 변환루틴을 써야 한다. 실례 8-13은 기본형과 사용자정의형사이의 변환을 보여준다. 이 실례에서 사용자정의형은 Distance클라스이고 기본형은 측정단위인 피트를 표시하는데 사용하는 float이다.

(실례 8-13) Distance를 피트로, 피트를 Distance로 변환하기

```
#include <iostream>
using namespace std;
class Distance
{
private:
   const float FTM;            // 피트를 메터로
   int meters; float centies;
public:
   Distance() : meters(0), centies(0), FTM(0.3038F) { }
   Distance(float feets) : FTM(0.3038F)
   {
      float fltMeters = FTM * feets;
      meters = int(fltMeters);
      centies = 100 * (fltMeters - meters);
   }
   Distance(int me, float ce) : meters(me), centies(ce), FTM(0.3038F) { }
   void GetDist()
   {
      cout << "\n메터를 입력하십시오:"; cin >> meters;
      cout << "센치메터를 입력하십시오:"; cin >> centies;
   }
   void ShowDist() { cout << meters << "m " << centies << "cm"; }
   operator float() const       // 변환연산자
   {
      float fracMeters = centies / 100; fracMeters += static_cast<float>(meters);
      return fracMeters / FTM;
   }
};
int main()
{
   float fts;
   Distance dist1 = 2.35F;
   cout << "\ndist1="; dist1.ShowDist();
   fts = static_cast<float>(dist1);
   cout << "\ndist1=" << fts << "feet";
   Distance dist2(5, 10.25);
   fts = dist2; cout << "\ndist2=" << fts << "feet\n";
   // dist2 = fts;
```

```
    return 0;
}
```

main()에서는 우선 1인수구성자에 의하여 피트를 나타내는 float량(2.35)을 메터와 센치메터로 변환한다.

```
Distance dist1 = 2.35F;
```

그다음 반대로 다음 명령문

```
fts = static_cast<float>(dist1);
```

과

```
fts = dist2;
```

에 의해 Distance를 피트로 변환한다.

여기에 프로그람의 출력이 있다.

```
dist1=0m 71.393cm
dist1=2.35feet
dist2=16.7956feet
```

main()에서 단순대입명령문에 의하여 변환을 처리하는 방법을 보았다.

그러면 Distance성원함수의 리면에 무엇이 숨어있는가를 고찰하자.

사용자정의형으로부터 기본형에로의 변환은 기본형으로부터 사용자정의형에로의 변환과 다른 수법을 요구한다. 실례 8-13에서는 두가지 변환방법을 보여준다.

1) 기본형으로부터 사용자정의형에로의 변환

기본형(이 경우에 float)으로부터 사용자정의형(Distance)으로 변환하는데 1인수구성자를 사용한다. 이것을 흔히 변환구성자(conversion constructor)라고 한다.

```
Distance(float feets) : FTM(0.3038F)
{
    float fltMeters = FTM * feets;
    meters = int(fltMeters);
    centies = 100 * (fltMeters - meters);
}
```

변환구성자함수는 Distance형객체를 하나의 명령문에 의해 창조할 때 호출된다. 함수는 인수가 피트를 표시한다고 가정하고 인수를 메터와 센치메터로 변환하고 결과값을 객체에 대입한다. 이리하여 피트로부터 Distance에로의 변환은 명령문

```
Distance dist1 = 2.35F;
```

에 의해 객체의 창조와 함께 수행된다.

2) 사용자정의형으로부터 기본형에로의 변환

그러면 사용자정의형으로부터 기본형에로 어떻게 변환하는가?

이것은 변환연산자(conversion operator)를 창조하여 변환한다.

```
operator float() const
{
    float fracMeters = centies / 100;
    fracMeters += static_cast<float>(meters);
    return fracMeters / FTM;
```

```
}
```

이 연산자는 그것이 성원으로 되는 Distance객체의 값을 가지며 이것을 메터를 표시하는 float값으로 변환하고 그 값을 돌려준다.

이 연산자는 명시적강제형변환에 의해 호출된다.

```
fts = static_cast<float>(dist1);
```

또한 단순대입과 함께 호출된다.

```
fts = dist2;
```

두개의 명령문은 Distance객체를 피트와 등가한 float값으로 변환한다.

3) C문자렬과 String객체들사이의 변환

1인수구성자와 변환연산자를 사용하는 다른 실례가 있다. 실례 8-14는 실례 8-6의 String클라스에 대하여 조작한다.

(실례 8-14) 일반문자렬과 String클라스사이의 변환

```
#include <iostream>
using namespace std;
#include <string.h>
class String
{
private:
   enum { SZ = 80 };
   char str[SZ];
public:
   String() { str[0] = '\0'; }
   String(char s[]) { strcpy(str, s); }
   void Display() const { cout << str; }
   operator char*() { return (char*)str; }
};
int main()
{
   String s1;
   char xstr[] = "Abcde";
   s1 = xstr;
   s1.Display();
   String s2 = "Klmnop";
   cout << static_cast<char*>(s2);
   cout << endl;
   return 0;
}
```

1인수구성자는 일반문자렬(char배렬)을 String클라스의 객체로 변환한다.

```
String(char s[]) { strcpy(str, s); }
```

인수로 넘어온 C문자렬은 strcpy()서고함수에 의하여 새로 창조된 String객체에 있는 str자료성원에 복사된다.

이 변환은 String을 창조할 때 다음과 같이 적용된다.

```
String s2 = "Klmnop";
```

또는 다음과 같이 대입명령문에서 적용된다.

```
s1 = xstr;
```

여기서 s1은 String형, xstr는 C문자렬이다.

변환연산자는 String객체를 C문자렬로 변환하는데 쓰인다.

```
operator char*() { return (char*)str; }
```

이 식에서 별표 *는 지적자를 의미한다. 이것은 char에로의 지적자를 의미하며 char형배렬과 비슷하다. 이것은 C문자렬자료형을 지정하는 다른 하나의 방법이다.

변환연산자는 다음 명령문

```
cout << static_cast<char*>(s2);
```

에서 번역프로그람에 의해 사용된다. 여기서 s2변수는 재정의된 <<연산자에 공급된 인수이다. <<연산자가 사용자정의 String형을 모르므로 번역프로그람은 s2을 <<연산자가 아는 형으로 변환하는 방법을 찾는다. char*강제형변환에 의하여 변환하려는 형을 지정함으로써 String으로부터 C문자렬에로의 변환연산자 char*()함수를 찾아서 C문자렬을 생성하고 <<연산자에 보내여 표시한다.

실례 8-14의 출력은 다음과 같다.

```
Abcdeklmnop
```

이 실례에서는 변환이 대입명령문뿐아니라 다른 적당한 위치(<<와 같은 연산자 혹은 함수에 보내는 인수중에서)에서도 자동적으로 발생한다. 옳지 않은 형의 인수를 연산자나 함수에 공급하면 작성자가 정의한 변환에 의하여 받아들일수 있는 형의 인수로 변환된다.

String을 C문자렬로 변환하는데 다음과 같은 명시적인 대입명령문을 사용할수 없다.

```
xstr = s2;
```

C문자렬 xstr는 배렬이며 일반적으로 배렬에는 대입할수 없다.

3. 각이한 클라스의 객체들사이의 변환

그러면 서로 다른 사용자정의클라스의 객체들사이의 변환은 어떻게 진행하는가?

이미 기본형과 사용자정의형들사이의 변환에서 보여준것과 같은 두가지 방법을 두개의 사용자정의형들사이의 변환에 적용할수 있다. 즉 1인수구성자를 사용하거나 변환연산자를 사용할수 있다. 그 선택여부는 원천객체 혹은 목적객체의 어느 클라스선언안에 변환루틴을 넣으려고 하는가에 달려있다. 례를 들면 다음과 같이 가정한다.

```
objecta = objectb;
```

여기서 objecta는 클라스 A의 객체, objectb는 클라스 B의 객체이다. 즉 클라스 A(objecta가 값을 받아들이는 목적클라스)에 변환루틴을 배치하겠는가, 클라스 B(원천클라스)에 변환루틴을 배치하겠는가? 이 두가지 경우를 고찰하자.

1) 두 종류의 시간

다음의 실례프로그람은 두가지 시간표시형식 즉 12시간형식의 시간과 24시간형식

의 시간사이를 변환한다. 이러한 시간표시방법을 흔히 민용시간과 군용시간이라고 한다. Time12클라스는 민용시간을 나타내고 수자형시간과 렬차출발시간을 표시하는데 사용된다. 이 경우에 초는 필요없으므로 Time12은 시간(1～12), 분 그리고 "오전" 또는 "오후"만 사용한다.

Time24클라스는 비행기려행과 같은 더 정확한 응용을 위한것으로써 시간(0～23), 분, 초를 요구한다. 표 8-1은 그 차이를 보여준다.

표 8-1. 12시간과 24시간형식의 차이

12시간형식	24시간형식
오전 12:00 (정오)	00:00:00
오전 12:01	00:01:00
오전 1:00	01:00:00
오전 6:00	06:00:00
오전 11:59	11:59:00
오후 12:00 (자정)	12:00:00
오후 12:01	12:01:00
오후 6:00	18:00:00
오후 11:59	23:59:00

민용시간에서 오전 12시(정오)가 군용시간에서 0시이다. 민용시간 0시는 없다. (정오는 형식적으로 오전 12시이고 자정은 오후 12시이지만 수자표시에 쓰이지 않는다.)

2) 원천객체에 변환루틴의 배치

첫 실례는 원천클라스에 배치된 변환루틴을 보여준다. 변환루틴을 원천클라스에 배치할 때 일반적으로 변환연산자로 실현한다. 여기에 실례 8-15가 있다.

(실례 8-15) Time24의 연산자에 의하여 Time24로부터 Time12로 변환

```
#include <iostream>
#include <string>
using namespace std;
class Time12
{
private:
   bool pm;  // 오후이면 true, 오전이면 false
   int hrs;     // 1-12
   int mins;  // 0-59
public:
   Time12() : pm(true), hrs(0), mins(0) {}
   Time12(bool ap, int h, int m) : pm(ap), hrs(h), mins(m) {}
   void Display() const
   {
      string am_pm = pm ? "오후" : "오전";
      cout << am_pm;
```

```
    cout << hrs << ':';
    if(mins < 10) cout << '0';
    cout << mins << ' ';
  }
};
class Time24
{
private:
  int hours;      // 0-23
  int minutes;        // 0-59
  int seconds;        // 0-59
public:
  Time24() : hours(0), minutes(0), seconds(0) {}
  Time24(int h, int m, int s) : hours(h), minutes(m), seconds(s) {}
  void Display() const
  {
    if(hours < 10) cout << '0';
    cout << hours << ':';
    if(minutes < 10) cout << '0';
    cout << minutes << ':';
    if(seconds < 10) cout << '0';
    cout << seconds;
  }
  operator Time12() const;
};
Time24::operator Time12() const
{
  int hrs24 = hours;
  bool pm = hours < 12 ? false : true;
  int roundMins = seconds < 30 ? minutes : minutes + 1;
  if(roundMins == 60)
  {
    roundMins = 0; ++hrs24;
    if(hrs24 == 12 || hrs24 == 24)
        pm = (pm == true) ? false : true;
  }
  int hrs12 = (hrs24 < 13) ? hrs24 : hrs24 - 12;
  if(hrs12 == 0)
  {
    hrs12 = 12;
    pm = false;
  }
  return Time12(pm, hrs12, roundMins);
}
int main()
{
  int h, m, s;
  while(true)
```

```
    {
      cout << "24시간형식의 시간을 입력하시오:\n";
      cout << "  시(0-23)? "; cin >> h;
      if(h > 23) return (1);
      cout << "  분? "; cin >> m;
      cout << "  초? " ; cin >> s;
      Time24 t24(h, m, s);
      cout << "입력한 시간은 "; t24.Display();
      Time12 t12 = t24;
      cout << "\n12시간형식의 시간은: "; t12.Display();
      cout << "\n\n";
    }
    return 0;
  }
```

실례의 main()부분에서 Time24형객체 t24를 정의하고 사용자가 입력한 시, 분, 초값을 객체들에 써넣는다. 또한 Time12형의 객체 t12을 정의하고

```
Time12 t12 = t24;
```

에 의해 t24로 초기화한다.

이 객체들의 클라스는 서로 다르므로 대입은 변환을 동반하고 이 프로그람에서는 변환연산자가 Time24클라스의 성원으로 되여있다. 여기에 정의가 있다.

```
Time24::operator Time12() const
{
  int hrs24 = hours;
  bool pm = hours < 12 ? false : true;
  int roundMins = seconds < 30 ? minutes : minutes + 1;
  if(roundMins == 60)
  {
    roundMins = 0; ++hrs24;
    if(hrs24 == 12 || hrs24 == 24)
        pm = (pm == true) ? false : true;
  }
  int hrs12 = (hrs24 < 13) ? hrs24 : hrs24 - 12;
  if(hrs12 == 0)
  {
    hrs12 = 12;
    pm = false;
  }
  return Time12(pm, hrs12, roundMins);
}
```

이 함수는 그것이 성원인 객체를 Time12객체로 변환하여 돌려주고 main()은 그것을 t12에 대입한다. 여기에 프로그람의 출력이 있다.

```
24시간형식의 시간을 입력하시오:
  시(0-23)? 17
  분? 59
  초? 45
```

```
입력한 시간은 17:59:45
12시간형식의 시간은: 오후 6:00
```

둘째 값이 반올림되여 12시간형식의 오후 5:59으로부터 오후 6:00으로 된다. 23보다 큰 시간값을 입력하면 프로그람은 완료한다.

3) 목적객체에 변환루틴의 배치

변환루틴이 목적클라스안에 있을 때 변환과정을 고찰하자. 이 경우에는 1인수구성자를 사용한다. 그러나 목적클라스의 구성자가 원천클라스안의 자료를 호출하여 변환을 처리하여야 한다는 사실로부터 복잡해진다. Time24의 자료들 즉 hours, minutes, seconds는 비공개이므로 Time24안에 특수한 성원함수들을 제공하여 그것을 직접 호출하게 해야 한다. 이것들은 GetHrs(), GetMins(), GetSecs()이다.

(실례 8-16) Time12의 구성자에 의하여 Time24로부터 Time12로 변환

```
#include <iostream>
#include <string>
using namespace std;
class Time24
{
private:
  int hours;       // 0-23
  int minutes;         // 0-59
  int seconds;         // 0-59
public:
  Time24() : hours(0), minutes(0), seconds(0) {}
  Time24(int h, int m, int s) : hours(h), minutes(m), seconds(s) {}
  void Display() const
  {
    if(hours < 10) cout << '0';
        cout << hours << ':';
    if(minutes < 10) cout << '0';
        cout << minutes << ':';
    if(seconds < 10) cout << '0';
        cout << seconds;
  }
  int GetHrs() const { return hours; }
  int GetMins() const { return minutes; }
  int GetSecs() const { return seconds; }
};
class Time12
{
private:
  bool pm;   // pm->true, aam->false
  int hrs;     // 1-12
  int mins;   // 0-59
public:
  Time12() : pm(true), hrs(0), mins(0) {}
  Time12(Time24);
```

```
    Time12(bool ap, int h, int m) : pm(ap), hrs(h), mins(m) {}
    void Display() const
    {
        string am_pm = pm ? "오후" : "오전";
        cout << am_pm;
        cout << hrs << ':';
        if(mins < 10) cout << '0';
            cout << mins << ' ';
    }
};
Time12::Time12(Time24 t24)
{
    int hrs24 = t24.GetHrs();
    pm = t24.GetHrs() < 12 ? false : true;
    mins = (t24.GetSecs() < 30) ? t24.GetMins() : t24.GetMins() + 1;
    if(mins == 60)
    {
        mins = 0; ++hrs24;
        if(hrs24 == 12 || hrs24 == 24)
            pm = (pm == true) ? false : true;
    }
    hrs = (hrs24 < 13) ? hrs24 : hrs24 - 12;
    if(hrs == 0)
    {
        hrs = 12;
        pm = false;
    }
}
int main()
{
    int h, m, s;
    while(true)
    {
        cout << "24시간형식의 시간을 입력하시오:\n";
        cout << "  시(0-23)? "; cin >> h;
        if(h > 23) return (1);
            cout << "  분? "; cin >> m;
        cout << "  초? " ; cin >> s;
        Time24 t24(h, m, s);
        cout << "입력한 시간은 "; t24.Display();
        Time12 t12 = t24;
        cout << "\n12시간형식의 시간은: "; t12.Display(); cout << "\n\n";
    }
    return 0;
}
```

여기에 1인수구성자로 되여있는 Time12클라스에로의 변환루틴이 있다.

```
Time12::Time12(Time24 t24)
{
```

```
        int hrs24 = t24.GetHrs();
        pm = t24.GetHrs() < 12 ? false : true;
        mins = (t24.GetSecs() < 30) ? t24.GetMins() : t24.GetMins() + 1;
        if(mins == 60)
        {
           mins = 0; ++hrs24;
           if(hrs24 == 12 || hrs24 == 24)
               pm = (pm == true) ? false : true;
        }
        hrs = (hrs24 < 13) ? hrs24 : hrs24 - 12;
        if(hrs == 0)
        {
           hrs = 12;
           pm = false;
        }
     }
```

이 함수는 그것이 성원인 객체를 인수로 받아들여 객체를 Time24클라스형의 값으로 설정한다. 그것은 실례 8-15의 변환연산자와 같은 방법으로 동작한다. 다만 Time24객체안에서 GetHrs()와 류사한 함수를 사용하여 자료를 호출하는 경우는 없다.

실례 8-16의 main()부분은 실례 8-15와 같다.

또한 1인수구성자는 다음의 명령문

```
Time12 t12 = t24;
```

에 의해 Time24를 Time12로 변환할수 있다.

출력도 같다. 차이는 원천객체안에 있는 변환연산자가 아니라 목적객체안의 구성자에 의해 변환이 조종되는데 있다.

4. 변환의 사용방법

목적클라스에서 1인수구성자를 사용해야 할 때 반대로 원천클라스에 있는 변환연산자를 사용할수 있는가?

대체로 그 선택은 작성자에게 달려있다. 작성자가 클라스서고를 구입하면 그 원천코드를 호출하지 않는다. 변환에서 원천클라스의 객체를 사용한다면 목적클라스만 호출할수 있으므로 1인수구성자를 사용해야 한다. 서고클라스객체가 목적지라면 원천클라스의 변환연산자를 사용해야 한다.

5. 연산자재정의와 변환에서 주의사항

연산자재정의와 형변환은 완전히 새로운 언어를 창조할 기회를 준다.

a, b, c가 사용자정의클라스들이고 +를 재정의한다면 명령문

```
a = b + c;
```

는 a, b, c가 기본자료형의 변수일 때 수행하는것과 전혀 동작이 다를수 있다. 언어의 구성블로크를 재정의하는 능력은 프로그람을 직관적이고 읽기 쉽게 한다. 또한 역효과

를 일으켜 프로그람을 명백하지 않고 리해하기 힘들게 한다. 여기에 그 원칙이 있다.

① 류사한 의미를 사용하여야 한다.

재정의된 연산자는 기본자료형에서 수행하는것과 류사한 연산을 처리해야 한다. 즉 +기호가 덜기를 처리하도록 재정의할수도 있으나 이것은 프로그람을 복잡하게 만든다.

연산자재정의는 일정한 클라스의 객체들에 대하여 특별한 연산을 수행하는것으로 가정한다. 클라스 X에서 +연산자를 재정의하려고 한다면 X클라스의 두 객체를 더한 결과는 적어도 더하기와 비슷한 의미를 가져야 한다. 실례로 이 장에서는 Distance클라스의 +연산자재정의방법을 보았다. 두개 거리의 더하기는 명백히 의미가 있다. 또한 String클라스의 +연산자를 재정의하였다. 여기서는 두개 문자렬의 더하기를 어떤 문자렬뒤에 다른 문자렬을 련결하여 셋째 문자렬을 만드는것으로 해석하였다.

이것도 직관적으로 만족되는 해석이다. 그러나 많은 클라스들에서 객체들의 더하기는 의의가 없다. 실례로 종업원자료를 보관하는 Employee라는 클라스의 두 객체를 더하려고 하지 않는다.

② 류사한 문법을 사용하여야 한다.

기본자료형을 사용할 때처럼 연산자를 재정의해야 한다. 실례로 alpha와 beta가 기본형이라면 다음 명령문

```
alpha += beta;
```

에서 대입명령문은 alpha를 alpha와 beta의 합으로 설정한다. 이 연산자의 재정의판은 류사성을 가져야 한다. 이것은

```
alpha = alpha + beta;
```

와 같은 연산을 수행한다. 여기서 +연산자는 재정의되였다.

연산자의 문법적특성을 변경할수 없다. 실례로 2항연산자를 단항연산자로 또는 그와 반대로 재정의할수 없다.

③ 제한을 보여준다.

+연산자를 재정의한다면 프로그람에서 친숙되지 않은것, 실례로

```
a = b + c;
```

와 같은 명령문이 실제로 무엇을 의미하는가에 대하여 구체적으로 연구해야 한다.

재정의된 연산자수가 너무 늘어나고 또 그것들이 비직관적인 방법들에서 쓰인다면 그것을 사용하게 되는 총적 목적을 잃어버릴것이고 프로그람은 더욱더 읽기 힘들어진다.

재정의된 연산자는 용도가 명백할 때에만 사용해야 한다. 그렇지 않으면 함수이름이 그 목적을 표시할수 있으므로 재정의된 연산자대신 함수를 사용하여야 한다. 실례로 문자렬의 왼변을 찾기 위한 함수를 쓰려고 한다면 &&와 같은 연산자를 재정의하기보다도 GetLeft()라고 하는것이 더 좋다.

④ 모호성을 피하여야 한다.

같은 변환(가령 Time24을 Time12로)을 처리하는데 1인수구성자와 변환연산자를 둘다 사용한다고 가정하자.

그러면 번역프로그람이 변환방법을 어떻게 알아내는가?

걱정할 필요는 없다. 번역프로그람은 무엇을 할지 모르는 상황에 놓이는것을 좋아하지 않으며 그러한 경우에는 오유를 경고한다. 따라서 한가지 이상의 방법으로 같은 변환을 하지 말아야 한다.

⑤ 모든 연산자들을 재정의할수는 없다.

다음과 같은 연산자들은 재정의할수 없다.

성원호출 또는 점연산자(.), 범위해결연산자(::), 조건연산자(?:), 성원지적(pointer-to-member)연산자(->).

*&와 같은 새로운 연산자를 창조하거나 재정의할수도 없다. 오직 현존 연산자만 재정의할수 있다.

제 4 절. 예약어 explicit와 mutable

일반적으로 사용하지 않는 두개의 예약어 explicit와 mutable을 고찰해보자. 이것들의 효과는 서로 다르다. 그러나 둘다 클라스성원들을 변경할수 있으므로 함께 취급한다. explicit예약어는 자료변환과 관련되지만 mutable은 더 미묘한 목적을 가진다.

1. explicit에 의한 변환의 방지

실례 8-15와 실례 8-16에서 본것처럼 변환연산자와 1인수구성자를 적당히 설치하여 변환을 가능하게 할수 있다.

그러나 다른 변환을 발생시키지 않으려고 할수도 있으며 또한 바라지 않는 변환은 실제로 제거해야 한다. 변환연산자가 처리하는 변환을 방지하는것은 간단하다. 연산자를 정의하지 않으면 된다. 그러나 구성자의 경우에는 간단하지 않다. 다른 형의 단일값을 가지는 1인수구성자에 의하여 객체를 구성하려고 하는데 일부 다른 경우에 암시적변환을 요구하지 않을수 있다. 이때 어떻게 하겠는가?

표준 C++는 예약어 explicit를 받아들여 이 문제를 해결한다.

explicit는 1인수구성자의 선언앞에 배치된다. 실례 8-17은 이것을 보여준다.

(실례 8-17) explicit에 의한 변환의 방지

```
#include <iostream>
using namespace std;
class Distance
{
private:
   const float FTM;
   int meters;
```

```
    float centies;
public:
    Distance() : meters(0), centies(0), FTM(0.3038F) { }
    explicit Distance(float feets) : FTM(0.3038F)
    {
        float fltMeters = FTM * feets;
        meters = int(fltMeters);
        centies = 100 * (fltMeters - meters);
    }
    void ShowDist() { cout << meters << "m " << centies << "cm"; }
};

void FancyDist(Distance);
int main()
{
    Distance dist1(2.35F);
//Distance dist1 = 2.35F;      // ctor가 암시적이므로 오유
    cout << "\ndist1="; dist1.ShowDist();
    float fts = 5.0F;
    cout << "\ndist1 ";
//FancyDist(fts);            // ctor가 암시적이므로 오유
    return 0;
}

void FancyDist(Distance d)
{
    cout << "메터와 센치메터 = ";
    d.ShowDist();
    cout << endl;
}
```

이 프로그람에는 메터와 센치메터를 출력하기 전에 문자렬 "메터와 센치메터 = "를 출력하여 Distance객체의 출력을 장식하는 함수 FancyDist()가 있다. FancyDist()함수의 인수는 Distance형이고 그러한 형의 변수를 넘기여 FancyDist()를 문제없이 호출할수 있다. 그런데 float형의 변수를 인수로 하여 다음과 같이

```
FancyDist(fts);
```

FancyDest()를 호출할수 있으므로 그것을 방지할 대책을 취하여야 한다.

번역프로그람은 함수의 호출에 맞게 연산자변환을 시도한다.

float형을 인수로 가지는 Distance구성자를 찾으면 이 구성자가 float를 Distance로 변환하고 함수에 Distance값을 넘기게 한다. 이것은 암시적변환이며 그것이 가능하게 해서는 안된다.

구성자를 explicit로 하면 암시적변환을 막을수 있다. 프로그람에서 FancyDist()호출로부터 설명문기호를 삭제하여 이것을 번역하면 번역프로그람은 변환을 처리할수 없다고 경고한다. explicit예약어가 없으면 이 호출에는 오유가 없다.

explicit구성자의 부작용에 의하여 같기기호를 사용하는 다음과 같은 객체초기화형식

Distance dist1 = 2.35F;

을 사용할수 없다.

반면에 괄호가 있는 경우 즉

Distance dist1(2.35F);

는 항상 동작한다.

2. mutable에 의한 const객체의 자료변경

일반적으로 const객체를 창조하여 객체의 어떤 성원도 변경하지 못하게 담보한다.

그런데 어떤 특정한 성원자료항목만 변경해야 하는 const객체를 만들어야 하는 경우가 있다.

실례로 Windows프로그람이 화면우에 그리는 창문을 고찰해보자. 창문에는 스크롤띠, 안내와 같이 창문에 소유된 일부 기능들도 가질수 있다. 일반적으로 여러가지 프로그람작성에 소유자관계를 사용하며 한 객체가 다른 객체의 한 속성일 때에는 더 큰 독립성을 가리킨다.

이러한 경우에 객체를 변경하지 않고 그 소유자만 변경할수 있다. 스크롤띠는 같은 크기, 색, 방향을 가지지만 그 소유자를 한 창문으로부터 다른 창문으로 변경할수 있다.

속성을 변경할수 없는 const스크롤띠를 창조하려고 하는데 그 소유자만을 변경해야 할 때 mutable예약어가 요구된다. 실례 8-18에서 이것을 보여준다.

(실례 8-18) mutable을 사용하여 const객체의 자료변경

```
#include <iostream>
#include <string>
using namespace std;
class Scrollbar
{
private:
   int size;
   mutable string owner;
public:
   Scrollbar(int sz, string own) : size(sz), owner(own) {}
   void SetSize(int sz) { size = sz; }
   void SetOwner(string own) const { owner = own; }
   int GetSize() const { return size; }
   string GetOwner() const { return owner; }
};
int main()
{
   const Scrollbar sBar(60, "창문1");
//sBar.SetSize(100);
```

```
    sBar.SetOwner("창문2");
    cout << sBar.GetSize() << ", " << sBar.GetOwner() << endl;
    return 0;
  }
```

size속성은 const객체안에서 변경할수 없는 각종 스크롤띠를 표시한다. 그러나 owner속성은 객체가 const라도 변경할수 있다. 이것을 사용하려면 mutable로 한다. main()에서는 const객체 sbar를 창조한다. 그 크기를 변경할수 없지만 소유자는 SetOwner()함수를 사용하여 변경할수 있다. 이러한 경우에 sbar는 론리적상수성을 가진다고 말한다. 이것은 리론적으로 변경할수 없으나 실천에서 제한된 방법으로 변경할수 있다는것을 의미한다.

요 약

이 장에서는 사용자정의자료형에 적용할수 있도록 보통의 C++연산자들에 새로운 의미를 부여하는 방법을 보았다. 예약어 operator는 연산자를 재정의하는데 사용되고 결과로 생기는 연산자는 작성자에 의해 제공된 의미를 받아들인다.

형변환은 연산자재정의와 밀접히 련관되여있다. 일부 변환은 사용자정의형과 기본형사이에서 발생한다. 그러한 변환에는 두가지 방법을 사용한다. 즉 1인수구성자는 기본형을 사용자정의형으로 변환하고 변환연산자는 사용자정의형을 기본형으로 변환한다. 어떤 사용자정의형을 다른 형으로 변환할 때에도 이 수법을 사용할수 있다.

표 8-2는 변환을 요약한다.

표 8-2. 형변환

	목적지안의 루틴	원천안의 루틴
기본형을 기본형으로	(내부형변환연산자)	
기본형을 클라스로	구성자	무효
클라스를 기본형으로	무효	변환연산자
클라스를 클라스로	구성자	변환연산자

예약어 explicit가 지정된 구성자는 암시적자료변환에서 사용할수 없다.

예약어 mutable이 지정된 자료성원은 그 객체가 const라도 변경할수 있다.

문 제

1. 연산자재정의는

① C++연산자들이 객체들과 작업하게 한다.

② C++가 조종할수 있는 그 이상의 연산자를 준다.

③ 현존 C++연산자에 자체의 새로운 의미를 준다.

④ C++연산자를 새로 만들게 한다.

어느것이 옳은가?

2. 클라스 X가 재정의된 연산자를 사용하지 않는다고 가정하고 클라스 X의 객체 x1를 같은 형의 객체 x2로부터 덜고 결과를 x3에 넣는 명령문을 쓰시오.

3. 클라스 X가 -연산자재정의를 가지고있다고 가정하고 문제 2에서와 같은 조작을 하는 명령문을 쓰시오.

4. >=연산자를 재정의할수 있는가?

5. 실례 8-1의 Counter클라스에 대하여 count를 증가시키지 않고 감소시키는 재정의된 연산자의 정의를 쓰시오.

6. 재정의된 단항연산자의 정의에 몇개의 인수가 요구되는가?

7. obj1, obj2, obj3이 클라스 C의 객체라고 하자. 명령문 obj3 = obj2 - obj1;가 정확히 동작하려면 재정의된 -연산자가

① 두개의 인수를 가져야 한다.

② 값을 돌려주어야 한다.

③ 이름없는 림시객체를 창조하여야 한다.

④ 연산수인 객체의 성원을 사용해야 한다.

어느것이 옳은가?

8. 실례 8-5의 Distance클라스의 재정의된 ++연산자의 정의를 쓰시오. 이 연산자는 meters성원자료에 1을 더할 때 다음 명령문이 가능하게 해야 한다.

```
dist1++;
```

9. 다음 명령문

```
dist2 = dist1++;
```

이 가능하도록 문제 8의 ++연산자를 재정의하시오.

10. 앞붙이형식을 사용할 때 뒤붙이형식을 사용할 때의 재정의된 ++연산자의 동작과 무엇이 다른가?

11. 두개의 문자렬객체를 더하는 방법을 서술하는 두개의 선언이 있다.

```
void Add(String s1, String s2)
String operator+(String s)
```

제2선언자와 제1선언자의 항목들의 대응관계를 말하시오.

① 제1선언자의 함수이름(Add)는 제2선언자의 ____과 대응된다.

② 제1선언자의 돌림값(void)는 제2선언자의 ____과 대응된다.

③ 제1선언자의 제1인수(s1)는 제2선언자의 ____과 대응된다.

④ 제1선언자의 제2인수(s2)은 제2선언자의 ____과 대응된다.

⑤ 제1함수가 성원으로 되는 객체는 제2함수의 ____과 대응된다.

대응되는 항목들은 다음과 같다.

ㄱ) 인수(s)

ㄴ) 그 연산자가 성원인 객체

ㄷ) 연산자(+)

ㄹ) 돌림값(String형)

ㅁ) 이 항목과 일치하지 않는다.

12. 재정의된 연산자는 항상 연산수개수보다 하나 적은 인수를 요구한다. 옳은가?

13. 산수대입연산자를 재정의할 때 결과는

① 연산자의 오른변에 있는 객체에로 간다.

② 연산자의 왼변에 있는 객체에로 간다.

③ 연산자의 성원인 객체에로 간다.

④ 돌려주어야 한다.

어느것이 옳은가?

14. 실례 8-6의 String클라스와 작업하고 연산수를 대문자로 변경하는 재정의된 ++연산자의 정의를 쓰시오. 서고함수 toupper()(CCTYPE)를 사용할수 있다.

15. 사용자정의클라스를 기본형으로 변환하기 위하여

① 기본(내부)변환연산자

② 1인수구성자

③ 재정의된 =연산자

④ 클라스의 성원인 변환연산자

를 정의할수 있다. 어느것이 옳은가?

16. 명령문 objA = objB;는 이 객체들이 서로 다른 형일 때 번역프로그람오유를 일으킨다. 옳은가?

17. 기본형을 사용자정의클라스로 변환하기 위하여

① 기본(내부)변환연산자

② 1인수구성자

③ 재정의된 =연산자

④ 클라스의 성원인 변환연산자

를 정의할수 있다. 어느것이 옳은가?

18. AClass obj = intVar;와 같은 정의를 조종하는 구성자를 정의하였다면 명령문 obj = intVar; 를 쓸수 있는가?

19. objA가 클라스 A형, objB가 클라스 B형, objA = objB;를 가능하게 하고 클라스 A로 가도록 변환루틴을 요구한다면 어떤 형의 변환루틴을 사용할수 있는가?

20. *연산자가 나누기를 처리하도록 재정의하는 객체를 번역프로그람이 요구하지 않는다. 옳은가?

련습문제

1. 실례 8-5의 Distance클라스에 두 거리를 더는 재정의된 연산자 -를 추가하시오. 이것은

```
dist3 = dist1 - dist2;
```

을 가능하게 한다. 연산자가 작은 수로부터 큰 수를 더는데 절대로 사용되지 않는다고 가정하시오.

2. 실례 8-6에서 재정의된 +연산자를 대신하는 재정의된 +=연산자를 추가하시오. 이 연산자는 다음의 명령문을 가능하게 한다.

```
s1 += s2;
```

여기서 s2은 s1에 더해지고 결과는 s1에 남아있다. 또한 연산자는 다음 계산에서 연산결과를 사용해야 한다. 즉

```
s3 = s1 += s2;
```

3. 6장의 련습 3의 클라스 Time을 변경하여 AddTime()함수대신 재정의된 +연산자에 의하여 두개의 시간을 더하시오. Time클라스를 시험하는 프로그람을 작성하시오.

4. 6장 련습 1에 기초하는 클라스 Int를 창조하시오. 4개의 옹근수산수연산자 (+, -, *, /)를 재정의하여 Int형객체들에 대하여 연산을 진행하시오. 그러한 산수연산의 결과가 int의 표준범위(-2,147,483,648～2,147,483,648)를 초과하면 연산자는 경고를 출력하고 프로그람을 완료하시오. 이 자료형은 산수자리넘침에 의해 발생하는 오유를 받아들이지 않게 하는데 사용할수 있다. 자리넘침검사를 쉽게 하기 위하여 long double형을 사용하시오. 이 클라스를 시험하는 프로그람을 작성하시오.

5. 련습 3에서 언급한 Time클라스에 앞붙이와 뒤붙이형식의 증가(++)와 감소(--) 연산자를 재정의하시오. 이 연산자들을 시험하는 프로그람을 작성하시오.

6. 련습 5의 Time클라스에 재정의된 -연산자를 사용하여 돌림값을 덜고 재정의된 *연산자에 의하여 float형의 수를 시간값에 곱하는 기능을 추가하시오.

7. 6장 련습 11의 4기능분수수산기에서 Fraction클라스를 변경하시오. 더하기, 덜기, 곱하기, 나누기에 재정의된 연산자들을 사용하시오. 또한 == 와 !=비교연산자를 재정의하고 두개의 분수값으로서 0/1, 0/1을 입력하면 순환을 끝내도록 하시오. LowTerm()함수를 최소항까지 약분한 인수값을 돌려주도록 변경하시오.

8. 7장 련습 12의 BMoney클라스를 변경하여 재정의된 연산자들로 다음의 산수연산을 하도록 하시오.

```
BMoney = BMoney + BMoney
BMoney = BMoney - BMoney
BMoney = BMoney * long double(화페량의 배수)
long double = BMoney / BMoney(총 화페량을 단가로 나누기)
BMoney = BMoney / long double (단가계산)
```

/연산자를 두번 재정의한다. 번역프로그람은 인수들이 차이나므로 두 연산자들을

구별할수 있다. BMoney객체들에 대한 산수연산에서 long double자료도 처리할수 있게 하시오. main()프로그람에서는 사용자가 두개의 화페문자렬과 류동소수점수를 입력할것을 요구하시오. 그다음 5가지 연산을 수행하고 결과를 표시하여야 한다. 이것은 사용자가 요구하는한 반복해야 한다. 일부 화페연산 실례로 BMoney * BMoney는 성립하지 않는다. 그리고 BMoney를 long double에 더할수 없다. 그러한 조작을 할수 없도록 하기 위해 BMoney를 long double로, 또는 long double을 BMoney로 변환하는 연산자를 포함하지 말아야 한다. 실례로 다음과 같은 식

```
bMon2 = bMon1 + w;
```

을 쓴다면 번역프로그람은 w를 BMoney로 자동변환하고 연산한다. 변환이 없으면 번역프로그람은 오유를 경고한다. 또한 필요한 변환구성자를 explicit로 하시오. 재정의된 연산자로 처리하는 방법을 모르는 화페연산이 일부 있다. 그것들은 연산자의 왼변이 아니라 오른변에 객체를 요구한다. 즉

```
long double * BMoney// 불가능
long double / BMoney// 불가능
```

9. 실례 8-12의 SafeArray클라스를 배렬의 상한과 하한(실례로 100～200)을 둘다 지원할수 있게 수정하시오. 재정의된 첨수연산자는 배렬이 호출될 때마다 한계를 벗어나지 않는가를 검사해야 한다. 상한과 하한을 지정하는 2인수구성자를 추가해야 한다. 동적기억할당을 설명하지 않았으므로 성원자료는 여전히 0으로 시작하여 99로 끝나는 배렬이지만 SafeArray용의 첨수들은 실제의 int배렬에서의 다른 첨수로 넘길수 있다. 실례로 사용자가 100～171범위를 선택한다면 이것을 arr[0]으로부터 arr[71]로 넘길수 있다.

10. 평면우의 점들을 극자리표계(반경과 각도)로 표시하는 클라스 Polar를 창조하시오. 두개의 Polar량을 더하는 +연산자를 재정의하시오. 평면에서 두 점의 더하기란 그 점들의 x자리표와 y자리표들사이의 더하기이다. 이것은 합의 x와 y자리표로 된다. 두개의 극자리표를 평면자리표로 변환하고 그것들을 더하고 결과로 생기는 평면자리표를 극자리표로 변환해야 한다.

11. 2장 련습 10과 5장 련습 11의 GArea구조체를 정보(int), 평(float)의 자료항목을 가지는 클라스로 변경하시오. 다음과 같은 성원함수들을 창조하시오.

- 인수없는 구성자
- 1인수구성자(double의 총 평수)
- 2인수구성자
- 사용자로부터 20:2000형의 정보, 평을 얻는 Get()
- 토지의 량을 같은 형식으로 출력하는 Show()
- 재정의된 +연산자 (GArea + GArea)
- 재정의된 -연산자 (GArea - GArea)
- 재정의된 *연산자 (GArea * double)

· 재정의된 /연산자 (GArea / GArea)

· 재정의된 /연산자 (GArea / double)

· operator double (double로 변환)

산수연산을 처리하기 위하여 매 객체의 자료를 개별적으로 더한다. 두개의 GArea 객체를 double형으로 변환하고 double형에 대하여 연산하여 GArea형으로 역변환할 때 변환연산자를 사용하시오. 재정의된 +연산자는

```
GArea GArea::operator+(GArea g2)
{
    return GArea(double(GArea(jong, pyong)) + double(s2));
}
```

12. 련습 8의 BMoney클라스와 련습 11의 GArea클라스를 사용하는 프로그람을 작성하시오. BMoney와 GArea사이를 변환하는 연산자를 쓰시오. 이때 한정보는 30000원으로 가정하시오. 사용자가 토지량과 화폐량을 각각 입력하면 그것을 다른 량으로 환산하여 결과를 표시하시오. 현재의 BMoney와 GArea클라스를 변경하시오.

제 9 장. 계승

계승은 객체지향프로그람작성법의 가장 강력한 특성이다. 계승은 현존클라스 또는 기초클라스로부터 파생클라스라는 새로운 클라스를 창조하는 과정이다. 파생클라스는 기초클라스의 모든 특성을 계승하는것과 함께 자체의 고유한 특성을 추가적으로 가질수 있다. 계승과정에 기초클라스는 변경되지 않는다. 계승관계를 그림 9-1에 주었다.

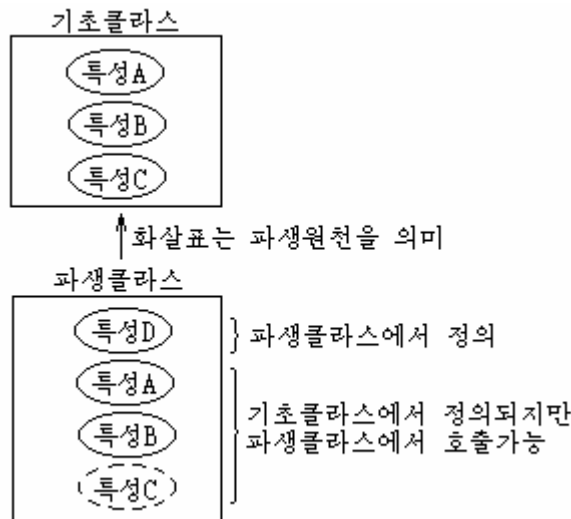



그림 9-1. 계승

그림 9-1에서 화살표는 기대와 달리 반대방향을 가리킨다. 아래로 가리키여 그것을 계승으로 표시할수 있다. 그러나 일반적으로 화살표는 우로 즉 파생클라스로부터 기초클라스에로 파생원천을 가리키게 한다.

계승은 객체지향프로그람작성법의 본질적인 부분이다. 계승의 가장 중요한 의의는 코드를 재리용할수 있게 한다는데 있다.

기초클라스를 작성하고 오유수정한 다음 그것을 다시 변경할 필요는 없지만 계승에 의하여 다른 상황에서 작업하도록 할수 있다. 현존코드의 재리용은 시간을 절약하고 프로그람의 믿음성을 제고한다. 또한 계승은 프로그람작성문제의 원시적개념화에 도움을 주고 프로그람의 전반설계를 방조한다.

재리용성의 중요한 의의는 클라스서고의 배포가 간단한것이다. 작성자는 다른 사람이나 회사에서 창조한 클라스를 변경하지 않고 사용할수 있으며 그로부터 특수한 정황에 알맞는 다른 클라스를 파생시킬수 있다.

이 장에서는 우선 계승이 필요한 리유를 설명하고 다음으로 기초클라스와 파생클라스, 호출조종, 클라스계층, 다중계승, 계승과 프로그람개발에 대하여 설명한다.

제 1 절. 파생클라스와 기초클라스

실례 8-3을 고찰하자. 여기서는 클라스 Counter를 일반계수기로 사용하였다. count는 0 또는 구성자에 주어진 값으로 초기화되고 ++연산자에 의해 증가되며 GetCount()로 얻을수 있다.

너무 오래 작업하여 Counter클라스를 요구대로 조작하기 힘들고 계수기가 너무 증가하여 계수기를 감소시키는 방법이 요구된다고 하자.

즉 은행에 들어오는 손님들을 계수하는 경우 그들이 들어오면 계수기를 증가, 밖으로 나가면 계수기를 감소시켜 계수기가 어느 순간에 은행안에 있는 손님수를 표시하게 하려고 한다.

Counter클라스의 원천코드에 직접 감소루틴을 삽입할수 있다. 그러나 그렇게 하지 말아야 할 리유가 있다. 우선 Counter클라스는 아주 잘 동작하고 이미 그 시험과 오유수정에 많은 시간을 소비하였다.

Counter의 원천코드를 변경하기 시작하면 시험을 다시 해야 하는것은 물론 어떤 오유를 범할수도 있으며 변경하기 전에 잘 동작하던 코드의 오유수정에 시간을 소비할수 있다.

Counter클라스를 변경하지 말아야 할 다른 리유가 있다. 실례로 클라스서고로서 배포되면 그 원천코드를 호출할수 없다.

이러한 문제를 피하기 위하여 계승을 사용하여 Counter를 변경하지 않고 그것에 기초하는 새로운 클라스를 창조한다. 여기에 새로운 클라스 CountDn을 포함하는 실례 9-1의 프로그람이 있다. 여기서는 CounterDn클라스에 감소연산자를 추가한다.

(실례 9-1) Counter클라스의 계승

```
#include <iostream>
using namespace std;
class Counter
{
protected:
   unsigned int count;
public:
   Counter() : count(0) {}
   Counter(int c) : count(c) {}
   unsigned int GetCount() { return count; }
   Counter operator++() { return Counter(++count); }
};
class CountDn : public Counter
{
public:
   Counter operator--() { return Counter(--count); }
};
```

```
int main()
{
   CountDn c1;
   cout << "\nc1=" << c1.GetCount();
   ++c1; ++c1; ++c1;
   cout << "\nc1=" << c1.GetCount();
   --c1; --c1;
   cout << "\nc1=" << c1.GetCount();
   cout << endl;
   return 0;
}
```

프로그람은 Counter클라스로 시작되는데 실례 8-3으로부터 변경되지 않았다. 간단하게 하기 위하여 뒤붙이식 ++연산자를 포함하지 않았다.

1. 파생클라스의 지정

프로그람에는 Counter클라스뒤에 새로운 클라스 CountDn의 지정이 있고 이 클라스에는 새로운 함수 operator--()(계수기감소)가 있다. 그런데 여기서 중요한것은 새로운 클라스 CountDn가 Counter클라스의 특성들을 모두 계승한다는데 있다. CountDn은 구성자와 GetCount() 혹은 operator++()가 이미 Counter에 있으므로 그것들을 정의하지 않는다.

CountDn의 첫 행에서는 그것이 Counter로부터 파생된다는것을 지정한다.

```
class CountDn:public Counter
```

여기서는 두 점과 public예약어, 기초클라스의 이름 Counter를 사용한다. 이것은 클라스들사이의 관계를 설정한다. 이 행은 CountDn이 기초클라스 Counter로부터 파생된다는것을 말해준다. 그 관계를 그림 9-2에서 보여준다.

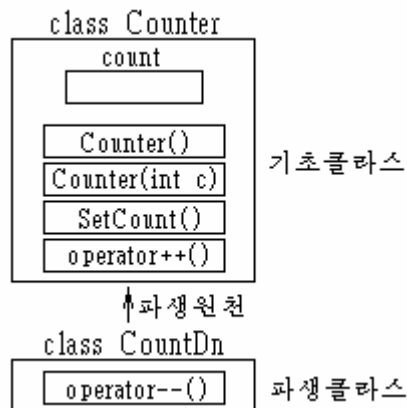



그림 9-2. 실례 9-1에서 클라스계승

그림에서 화살표는 파생원천을 의미한다. 화살표는 파생클라스가 기초클라스안의 함수와 자료들을 참고하고 기초클라스가 파생클라스에로 호출할수 없다는것을 강조한다. 이런 그라프를 방향있는 순환그라프라고 하며 또한 계승이라고도 한다.

2. 기초클라스성원의 호출

계승에서 중요한것은 파생클라스의 객체들이 기초클라스의 성원함수를 언제 사용하는가를 알고있는것이다. 이것을 호출가능성(accessibility)이라고 한다.

그러면 번역프로그람이 실례 9-1에서 호출가능성을 어떻게 조절하는가를 고찰하자.

1) 기초클라스구성자의 대용

실례 9-1의 main()에서 클라스 CountDn의 객체를 창조한다.

```
CountDn  c1;
```

여기서 c1은 클라스 CountDn의 객체로 창조되고 0으로 초기화된다.

그러면 이것이 어떻게 동작하는가? CountDn에 구성자가 없는데 어떻게 실체를 초기화하는가?

일정한 환경에서 구성자를 지정하지 않으면 파생클라스는 기초클라스로부터 적당한 구성자를 사용한다. 실례 9-1에서는 CountDn에 구성자가 없으므로 번역프로그람은 Counter로부터 인수없는 구성자를 사용한다.

2) 기초클라스성원함수의 대용

또한 CountDn클라스의 객체 c1은 Counter클라스로부터 operator++()와 GetCount()를 사용한다.

우선 c1을 증가시킨다.

```
++c1;
```

그다음 c1의 count를 표시한다.

```
cout << "\nc1= " << c1.getCount();
```

번역프로그람은 c1이 성원인 클라스에서 이 함수들을 찾지 못하고 기초클라스의 성원함수들을 사용한다.

3) 실례 9-1의 출력

main()에서는 c1을 세번 증가시키고 결과값을 출력하며 c1을 두번 감소시키고 마감에 그 값을 다시 출력한다. 출력은 다음과 같다.

```
c1=0
c1=3
c1=1
```

Counter클라스의 ++연산자, 구성자, GetCount()함수 그리고 CountDn클라스의 --연산자가 모두 CountDn형의 객체에 대하여 동작한다.

3. protected호출지정자

클라스를 변경하지 않고 기능을 확장하였다. 그러면 Counter클라스에 대한 한가지 변경을 고찰해보자.

실례 8-3에서 Counter클라스에 포함된 count자료는 private호출지정자를 사용한다.

실례 9-1의 Counter클라스에서 count는 새로운 지정자 protected로 주어진다. 무엇때문인가?

먼저 호출지정자 private와 public를 상기해보자.

클라스성원(자료 혹은 함수)들은 항상 자기 클라스안에 있는 함수에 의해 호출된다. 그것들은 private 혹은 public성원이다. 그러나 클라스밖에서 정의된 클라스의 객체는 그 성원이 public일 때에만 클라스성원들을 호출할수 있다. 실례로 객체 objA가 클라스 A의 실례이고 함수 FuncA()가 A의 성원함수라고 하자. main() 혹은 A의 성원이 아닌 다른 함수안에서 명령문

```
objA.FuncA();
```

는 FuncA()가 공개가 아니면 옳지 않다. 객체 objA는 클라스 A의 비공개성원을 호출할수 없다. 물론 비공개성원들은 private이다. 그림 9-3에 그것을 보여준다.

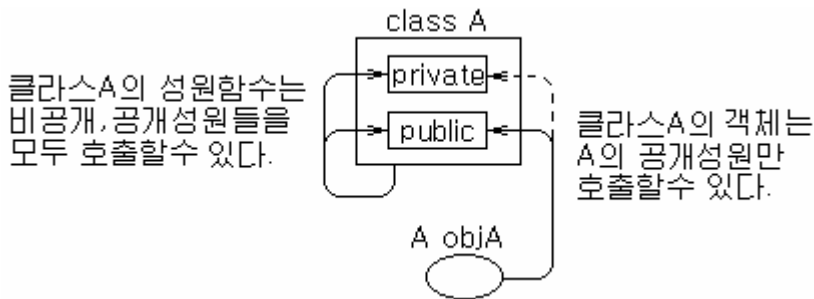



그림 9-3. 계승이 없을 때 호출지정자

이것은 계승을 사용하지 않는 경우에 늘 알고있어야 한다. 계승을 사용할 때에는 보충적인 가능성이 얼마든지 있다. 한가지 질문이 있다.

파생클라스의 성원함수들이 기초클라스의 성원을 호출할수 있는가? 다시말하여 CountDn안의 operator--()가 Counter안의 count를 호출할수 있는가?

기초클라스의 성원이 공개 혹은 보호라면 그 성원들을 호출할수 있다.

count를 프로그람의 어느 곳에서나 임의의 함수로부터 호출할수 있으면 자료은페의 우점을 잃어버릴수 있으므로 공개성원으로 하지 않는다. 또한 다른 보호성원은 클라스자체 또는 그 파생클라스의 성원함수들이 호출할수 있다. main()과 같은 클라스밖의 함수로부터는 호출할수 없다. 이러한 상황을 그림 9-4에서 보여준다. 또한 표 9-1에 각이한 방법들을 요약한다.

표 9-1 . 계승과 호출가능성

호출지정자	자기클라스로부터 호출가능	파생클라스로부터 호출가능	클라스밖의 객체로부터 호출가능
public	○	○	○
protected	○	○	×
private	○	×	×

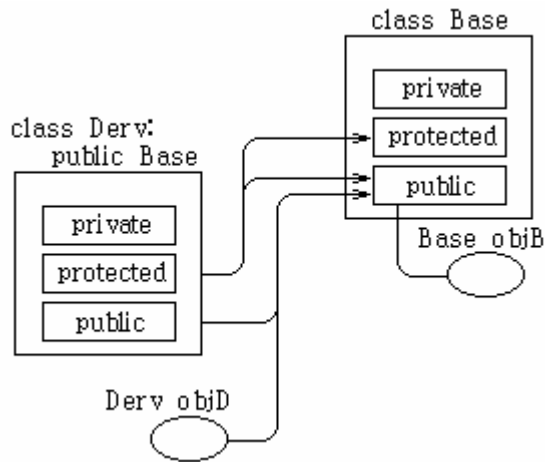



그림 9-4. 계승을 가지는 호출지정자

이상으로부터 앞으로 다른 클라스들의 기초클라스로서 사용하려는 클라스를 쓰는 경우에 파생클라스가 호출해야 할 자료를 공개성원이 아니라 보호성원으로 해야 한다는것을 알수 있다. 이것은 클라스를 계승할 준비가 되여있다는것을 담보한다.

1) protected의 위험성

클라스성원들을 보호성원으로 할 때 결함이 있다. 공개적으로 배포할 클라스서고를 쓴다고 하자.

이 서고를 구입하는 사람은 그것으로부터 단순히 다른 클라스를 파생시키는 방법으로 클라스의 보호성원들을 호출할수 있다. 이것은 비공개성원보다도 보호성원의 안전성이 상당히 낮다는것을 의미한다. 자료변경을 피하기 위하여 보통 main()프로그람이 하는것처럼 기초클라스안의 공개함수들만 사용하여 기초클라스의 자료를 파생클라스들이 호출하게 하는것이 더 안전하다. protected지정자의 사용은 프로그람작성을 더 단순화하므로 이 책의 실례들은 그것에 기초하고있다.

2) 변경되지 않는 기초클라스

기초클라스로부터 일부 클라스들이 파생되여도 기초클라스는 변하지 않는다.

실례 9-1의 main()부분에서는 Counter형의 객체로 정의할수 있다.

```
Counter c2;            // 기초클라스의 객체
```

이러한 객체는 CountDn이 존재하지 않는것처럼 동작한다.

또한 계승은 반대로 작용하지 않는다. 기초클라스와 그 객체들은 기초클라스로부터 파생된 클라스들에 대하여 전혀 모른다. 이 실례에서는 c2과 같은 Counter클라스의 객체가 CountDn의 operator--()함수를 사용할수 없다는것을 의미한다.

감소계수기를 요구한다면 그것은 Counter가 아니라 CountDn클라스의 객체이여야 한다.

3) 다른 용어들

일부 언어들에서 기초클라스를 상위클라스(superclass)라고 부르고 파생클라스를

보조클라스(subclass)라고 한다. 기초클라스를 부모클라스(parent class), 파생클라스를 자식클라스(child class)라고도 한다.

제 2 절. 파생클라스구성자와 성원함수의 재정의

1. 파생클라스구성자

실례 9-1에는 결함이 있다.

그러면 CounDn객체를 어떤 값으로 초기화한다면 어떤 일이 생기는가? 이때 Counter에서 1인수구성자를 사용할수 있는가? 없다.

실례 9-1에서 본것처럼 번역프로그람은 기초클라스로부터 인수없는 구성자를 대용하지만 더 복잡한 구성자들을 사용할수 없다. 이것을 가능하게 하자면 파생클라스에 새로운 구성자들을 추가해야 한다. 이것을 실례 9-2에서 보여준다.

(실례 9-2) 파생클라스의 구성자

```
#include <iostream>
using namespace std;
class Counter
{
protected:
   unsigned int count;
public:
   Counter() : count(0) {}
   Counter(int c) : count(c) {}
   unsigned int GetCount() { return count; }
   Counter operator++() { return Counter(++count); }
};
class CountDn : public Counter
{
public:
   CountDn() : Counter() {}
   CountDn(int c) : Counter(c) {}
   Counter operator--() { return Counter(--count); }
};
int main()
{
   CountDn c1;
   CountDn c2(100);
   cout << "\nc1=" << c1.GetCount();
   cout << "\nc2=" << c2.GetCount();
   ++c1; ++c1; ++c1;
   cout << "\nc1=" << c1.GetCount();
   --c2; --c2;
   cout << "\nc2=" << c2.GetCount();
```

```
    CountDn c3 = --c2;
    cout << "\nc3=" << c3.GetCount();
    cout << endl;
    return 0;
}
```

이 프로그람은 CountDn클라스의 두개의 새로운 구성자들을 사용한다. 여기서 인수없는 구성자는 다음과 같이 정의된다.

```
CountDn() : Counter() {}
```

이 구성자는 아직 잘 모르는 값을 가진다. 이 구성자는 함수이름뒤에 두 점이 있고 CountDn()구성자가 기초클라스의 Counter()구성자를 호출하게 한다. main()에서

```
CountDn c1;
```

라고 할 때 번역프로그람은 CountDn형의 객체를 창조하고 CountDn구성자를 호출하여 그것을 초기화한다.

CountDn구성자는 Counter구성자를 호출한다. CountDn()구성자는 자체의 보충적인 명령문들을 가질수 있지만 이 경우에는 그렇게 하지 않고 함수본체는 빈괄호로 되여있다.

초기화자목록에 의한 구성자호출이 이상하게 보일수 있으나 일리가 있다. 파생클라스나 기초클라스의 구성자를 실행하기 전에 파생클라스 혹은 기초클라스에서 임의의 변수를 초기화해야 하는 경우가 있다. 파생클라스구성자가 실행을 시작하기 전에 기초클라스구성자를 호출하여 이것을 달성한다.

main()에서 명령문

```
CountDn c2(100);
```

은 CountDn의 1인수구성자를 사용한다. 또한 이 구성자는 기초클라스의 대응하는 1인수구성자를 호출한다. 즉

```
CountDn(int c) : Counter(c) {}      // 인수 c를 Counter에 넘긴다.
```

이 구성자는 인수 c를 CountDn()으로부터 Coiunter()에 넘기여 객체를 초기화하는데 사용된다.

main()에서 c1과 c2객체를 초기화한 다음 c1을 증가시키고 c2를 감소시키며 그 결과를 출력한다. 1인수구성자는 또한 대입명령문

```
CountDn c3 = --c2;
```

에서 사용된다.

2. 성원함수재정의

파생클라스에서 기초클라스의 성원들을 재정의하여 사용할수 있다. 기초와 파생의 두 클라스의 객체를 같은 방법으로 호출할수 있다.

여기에 실례 7-8에 기초한 실례가 있다. 이 프로그람은 단순자료보관장소인 탄창을 모형화한다. 즉 탄창에 옹근수를 밀어넣고 탄창에서 옹근수를 꺼낸다. 그러나 실례 7-8에는 결함이 있다. 탄창에 너무 많은 항목을 밀어넣으면 자료가 st[]배렬의 끝을

지나서 기억기에 보관되므로 프로그람이 폭주할수 있다. 또한 너무 많은 항목을 꺼내려고 하면 배렬밖의 기억위치로부터 자료를 읽어들이므로 무의미한 결과를 얻는다. 이러한 결함을 없애기 위하여 Stack로부터 새로운 파생클라스 Stack2를 창조한다. Stack2의 객체들은 Stack와 같은 방법으로 동작하지만 탄창에 너무 많은 항목을 밀어넣으려고 하거나 빈 탄창에서 항목을 꺼내려고 하는 경우에는 례외로 된다. 여기에 실례 9-3이 있다.

(실례 9-3) 기초와 파생클라스에서 함수의 재정의

```
#include <iostream>
using namespace std;
#include <process.h>
class Stack
{
protected:
   enum {MAX = 3};
   int st[MAX];
   int top;
public:
   Stack() { top = -1; }
   void Push(int var) { st[++top] = var; }
   int Pop() { return st[top--]; }
};
class Stack2 : public Stack
{
public:
   void Push(int var)
   {
      if(top >= MAX - 1)
      {
         cout << "\n오유: 탄창이 다 찼습니다.";
         exit(1);
      }
      Stack::Push(var);
   }
   int Pop()
   {
      if(top < 0)
      {
         cout << "\n오유: 탄창이 비였습니다.\n";
         exit(1);
      }
      return Stack::Pop();
   }
};
int main()
{
```

```
    Stack2 s1;
    s1.Push(11);
    s1.Push(22);
    s1.Push(33);
    cout << endl << s1.Pop();
    cout << endl << s1.Pop();
    cout << endl << s1.Pop();
    cout << endl << s1.Pop() << endl;
    return 0;
}
```

이 프로그람은 Stack클라스의 자료성원들을 보호로 만든것을 제외하면 실례 7-8과 같다.

Stack2클라스에는 두개의 함수 Push()와 Pop()가 있다. 이 함수들은 Stack의 함수들과 같은 이름과 인수, 돌림값형을 가진다. main()에서

```
s1.Push(11);
```

와 같이 함수를 호출할 때 번역프로그람은 두개의 Push()함수들중 어느것을 사용하는가 하는 규칙이 있다. 기초와 파생의 두 클라스에 같은 함수가 존재하면 파생클라스의 함수가 실행된다. (이것은 파생클라스의 객체인 경우이다. 기초클라스의 객체인 경우에는 파생클라스에 대하여 전혀 모르므로 늘 기초클라스의 함수를 사용한다.) 일반적으로 파생클라스의 함수가 기초클라스의 함수를 재정의(overriding)한다고 말한다. 우의 명령문에서 s1은 클라스 Stack2의 객체이므로 Stack가 아니라 Stack2의 Push()함수가 실행된다.

Stack2의 Push()함수는 탄창이 다 찼는가 검사하고 다 찼으면 오유를 통보하고 프로그람을 완료하며 그렇지 않으면 Stack의 Push()함수를 호출한다. 마찬가지로 Stack2의 Pop()함수는 탄창이 비였는가를 조사하고 비였으면 오유통보를 출력하고 완료하며 그렇지 않으면 Stack의 Pop()함수를 호출한다.

main()에서는 탄창에 세개의 항목을 밀어넣지만 4개의 항목을 꺼낸다. 그러므로 마지막 꺼내기는 오유통보를 표시한다.

```
33
22
11
오유: 탄창이 비였습니다.
```

그리고 프로그람을 완료한다.

- 재정의된 함수에서 범위해결

그러면 Stack2의 Push()와 Pop()가 Stack의 Push()와 Pop()를 어떻게 호출하는가? 명령문

```
Stack::Push(var);
```

와

```
return Stack::Pop();
```

에서는 범위해결연산자 ::을 사용한다.

이 명령문들은 Stack의 Push()와 Pop()를 호출하도록 지정한다. 범위해결연산자가 없으면 번역프로그람은 Stack2의 Push()와 Pop()함수들을 호출한다. 범위해결연산자를 사용하면 함수가 어느 클라스의 성원인가를 정확히 지정할수 있다.

3. Distance클라스에서 계승

계승의 복잡한 례를 고찰하자. 지금까지 Distance클라스를 사용하는 각종 프로그람들에서는 거리가 항상 정수를 표시하는것으로 가정하였다. 이것은 일반적인 거리의 경우이다. 그러나 측정에서는 즉 조선서해의 수위가 조류에 따라 변하므로 부수의 메터나 센치메터량으로 표시해야 할 필요가 제기된다.(조류의 수위가 평균보다 아래인 경우 미누스조류라고 한다.)

Distance에서 새로운 클라스를 파생시킨다. 파생클라스에는 메터와 센치메터량에 부 또는 정을 나타내는 부호를 하나의 자료항목으로 추가한다. 부호를 더할 때에도 역시 부호있는 거리를 사용할수 있게 성원함수들을 변경할수 있다. 여기에 실례 9-4가 있다.

(실례 9-4) Distance를 사용한 계승

```
#include <iostream>
using namespace std;
enum PosNeg { POS, NEG };
class Distance
{
protected:
   int meters;
   float centies;
public:
   Distance() : meters(0), centies(0.0) { }
   Distance(int me, float ce) : meters(me), centies(ce) { }
   void GetDist()
   {
      cout << "\n메터를 입력하십시오:";
      cin >> meters;
      cout << "센치메터를 입력하십시오:";
      cin >> centies;
   }
   void ShowDist()    { cout << meters << "m " << centies << "cm"; }
};
class DistSign : public Distance
{
private:
   PosNeg sign;
public:
   DistSign() : Distance() { sign = POS; }
```

```
    DistSign(int me, float ce, PosNeg sg=POS) : Distance(me, ce)
        { sign = sg; }
    void GetDist()
    {
        Distance::GetDist();
        char ch;
        cout << "부호(+ 혹은 -)를 입력하십시오:";
        cin >> ch;
        sign = (ch == '+') ? POS : NEG;
    }
    void ShowDist()
    {
        cout << ((sign == POS) ? "+" : "-");
        Distance::ShowDist();
    }
};
int main()
{
    DistSign alpha;
    alpha.GetDist();
    DistSign beta(11, 6.25);
    DistSign gamma(100, 5.5, NEG);
    cout << "\nalpha=";
    alpha.ShowDist();
    cout << "\nbeta=";
    beta.ShowDist();
    cout << "\ngamma=";
    gamma.ShowDist();
    cout << endl;
    return 0;
}
```

여기서 DistSign클라스에는 부호있는 수를 취급하는 기능을 추가한다. 실례에서 Distance클라스는 앞의 프로그람과 같지만 자료가 보호라는것이 다르다. 파생클라스함수호출이 실제로 하나도 없는 경우에는 비공개로 할수 있지만 필요한 경우에는 파생클라스함수를 호출할수 있도록 보호로 만드는것이 더 좋다.

1) 실례 9-4의 연산

main()프로그람은 세개의 각이한 거리를 선언한다. 이 함수는 사용자로부터 alpha값을 얻고 beta를 +11m 6.25cm로, gamma를 -100m 5.5cm로 초기화한다. 출력은 다음과 같다.

```
메터를 입력하십시오:6
센치메터를 입력하십시오:2.5
부호(+ 혹은 -)를 입력하십시오:-
alpha=-6m 2.5cm
beta=+11m 6.25cm
gamma=-100m 5.5cm
```

DistSign클라스는 Distance로부터 파생되고 PosNeg형의 sign을 유일한 값으로 가진다. sign변수는 거리의 부호를 보관한다. PosNeg형은 enum명령문에서 두개의 가능한 값 POS와 NEG를 가지도록 정의된다.

2) DistSign의 구성자

DistSign은 Distance를 반영하는 두개의 구성자를 가진다. 첫 구성자에는 인수가 없고 둘째 구성자는 두개 또는 세개의 인수를 가진다. 셋째 구성자는 둘째 구성자의 인수가 부호있을 때 POS 혹은 NEG를 선택적으로 가지며 기정값은 POS이다. 이 구성자들은 여러가지 방법으로 DistSign형의 변수(객체)를 정의할수 있다.

DistSign의 두개 구성자는 Distance에 대응하는 구성자를 호출하여 메터와 센치메터값들을 설정한 다음에 sign의 값을 설정한다. 인수없는 구성자는 늘 sign을 POS로 설정한다. 둘째 구성자는 sign을 제3인수가 제공되지 않으면 POS로, 인수가 지정되면 그 값(POS 또는 NEG)으로 설정한다.

DistSign의 둘째 구성자에 main()에서 넘긴 인수 me와 ce는 Distance의 구성자에로 넘어간다.

3) DistSign의 성원함수

Distance에 대한 부호의 추가는 두개의 성원함수에 영향을 준다. DistSign클라스의 GetDist()함수는 사용자로부터 메터와 센치메터값은 물론 부호를 얻어야 하고 ShowDist()함수는 메터와 센치메터를 부호와 함께 표시해야 한다. 이 함수들은 Distance에 대응하는 함수들을 호출한다. 즉

```
Distance::GetDist();
```

와

```
Distance::ShowDist();
```

이 호출에서 메터와 센치메터값을 얻고 표시한다. DistSign의 GetDist()와 ShowDist()의 본체들에서는 그다음 부호를 처리한다.

4) 계승의 사용

C++는 파생클라스를 창조하는데 편리하도록 설계되였다. 기초클라스의 일부를 사용하려고 한다면 그것이 자료인가, 구성자인가 또는 성원함수인가 하는데 관계없이 어느것이나 사용할수 있다. 그다음 새로 개량된 클라스를 창조할 때 필요한 기능들을 추가한다. 실례 9-4에서는 코드를 중복하지 않고 기초클라스의 적당한 함수를 사용한다.

제 3 절. 클라스계층

지금까지 이 장의 실례들에서는 계승을 현존클라스의 기능을 추가하는데 사용하였다. 계승을 다른 목적 즉 프로그람의 원시실례의 부분으로 사용하는 실례를 고찰하자.

실례로 기업소의 종업원자료기지를 모의한다. 여기서는 간단히 3부류의 종업원들

을 표시한다. Manager는 관리일군, Scientist는 더 좋은 제품을 개발하기 위한 연구를 진행하며 Labour는 위험한 형단조프레스를 조종한다.

자료기지에는 모든 종업원들의 이름과 식별번호를 보관한다. 종업원의 부류는 보관하지 않는다. 그러나 관리일군에 대해서는 직무를 보관한다. 과학자에 대해서는 그가 집필한 출판물의 건수를 보관한다. 로동자에게는 자기 이름과 번호외에 추가자료가 없다.

실례프로그람은 기초클라스 Employee로 시작한다. Employee클라스는 종업원의 이름과 번호를 포함한다. Employee로부터 세개의 클라스 Manager, Scientist, Labour를 파생시킨다. Manager와 scientist클라스는 그 부류의 종업원에 대한 추가정보와 그 정보를 조종하는 성원함수를 포함한다.

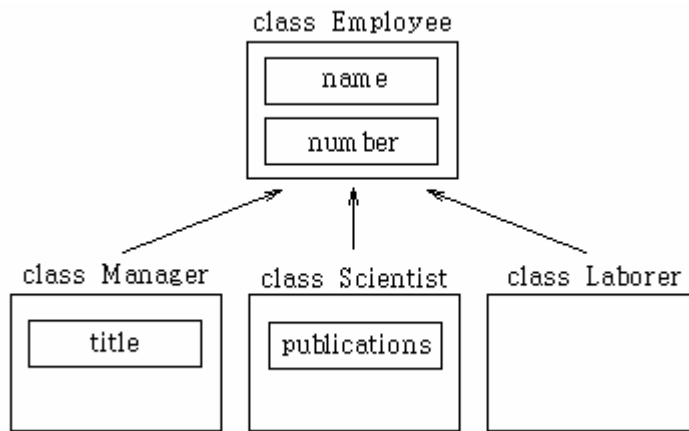



그림 9-5. 실례 9-5의 클라스계층

(실례 9-5) 계승을 사용한 종업원자료기지의 모의

```
#include <iostream>
using namespace std;
const int LEN = 80;
class Employee
{
private:
   char name[LEN];
   unsigned long number;
public:
   void GetData()
   {
      cout << "\n  이름? "; cin >> name;
      cout << "\n  종업원번호? "; cin >> number;
   }
   void PutData() const
   {
      cout << "\n  이름: " << name;
      cout << "\n  종업원번호: " << number;
```

```
    }
};
class Manager : public Employee
{
private:
    char title[LEN];
public:
    void GetData()
    {
        Employee::GetData();
        cout << "  직위? "; cin >> title;
    }
    void PutData() const
    {
        Employee::PutData();
        cout << "\n  직위: " << title;
    }
};
class Scientist : public Employee
{
private:
    int pubs;
public:
    void GetData()
    {
        Employee::GetData();
        cout << "  출판물건수? "; cin >> pubs;
    }
    void PutData() const
    {
        Employee::PutData();
        cout << "\n  출판물건수: " << pubs;
    }
};
class Laborer : public Employee { };
int main()
{
    Manager m1, m2;
    Scientist s1;
    Laborer l1;
    cout << endl;
    cout << "\n관리일군1의 자료를 입력하시오"; m1.GetData();
    cout << "\n관리일군2의 자료를 입력하시오"; m2.GetData();
    cout << "\n기사1의 자료를 입력하시오"; s1.GetData();
    cout << "\n로동자1의 자료를 입력하시오"; l1.GetData();
    cout << "\n관리일군1의 자료"; m1.PutData();
    cout << "\n관리일군2의 자료"; m2.PutData();
    cout << "\n기사1의 자료"; s1.PutData();
```

```
    cout << "\n로동자1의 자료"; l1.PutData();
    cout << endl;
    return 0;
  }
```

main()에서는 각이한 클라스의 4개 객체 즉 두개의 Manager, 하나의 Scientist, 하나의 Labour를 선언한다. 그다음 GetData()성원함수를 호출하여 매 종업원의 자료를 얻고 PutData()함수에 의하여 정보를 표시한다. 여기에 프로그람과의 간단한 대화가 있다. 우선 사용자가 자료를 입력한다.

```
관리일군1의 자료를 입력하시오
  이름? 김철호
  종업원 번호? 1
  직위? 지배인
관리일군2의 자료를 입력하시오
  이름? 리동호
  종업원 번호? 124
  직위? 회계과장
기사1의 자료를 입력하시오
  이름? 강철
  종업원 번호? 234
  출판물건수? 20
로동자1의 자료를 입력하시오
  이름? 최춘일
  종업원 번호? 400
```

이 프로그람의 실행결과는 다음과 같다.

```
관리일군1의 자료
  이름: 김철호
  종업원 번호: 1
  직위: 지배인
관리일군2의 자료
  이름: 리동호
  종업원 번호: 124
  직위: 회계과장
기사1의 자료
  이름: 강철
  종업원 번호: 234
  출판물건수: 20
로동자1의 자료
  이름: 최춘일
  종업원 번호: 400
```

복잡한 프로그람에서는 배렬이나 다른 용기를 사용하여 대량의 종업원객체를 취급할수 있게 자료를 배렬한다.

1. 추상기초클라스

실례에서는 기초클라스 Employe의 객체를 정의하지 않는다. 기초클라스

Employee는 다른 클라스를 파생시키는 기초로서 사용하자는데 그 목적이 있는 일반 클라스이다.

Labour클라스에는 자료나 함수가 추가되지 않으므로 그것은 Employee클라스처럼 작업한다. 이것은 Labour클라스가 필요없는것같아 보이지만 그것을 따로 분리하여 모든 클라스가 같은 원천인 Employee로부터 파생된다는것을 강조한다. 또한 앞으로 Employee의 선언을 변경하지 않고 Labour클라스를 변경한다.

실례의 Employee처럼 다른 클라스를 파생시키는데만 사용하는 클라스를 일반적으로 추상클라스(abstract class)라고 한다. 이것은 추상클라스의 실체의 실례(객체)를 창조하지 않는다는것을 의미한다.

2. 구성자와 성원함수

기초클라스와 파생클라스에 구성자가 없으므로 번역프로그람은 다음의 정의

```
Manager m1, m2;
```

를 만나는 경우에 Employee의 기정구성자를 호출하는 Manager의 기정구성자에 의하여 클라스의 객체들을 자동적으로 창조한다.

Employee의 GetData()와 PutData()함수는 사용자로부터 이름과 나이를 받아들이고 이름과 번호를 표시한다. Manager와 Scientist클라스에서 호출한 GetData()와 PutData()함수도 Employee의 함수들을 사용한 다음 자기의 작업도 진행한다. Manager에서 GetData()함수는 사용자로부터 title을 얻으며 PutData()는 그 값을 표시한다. Scientist에서 이 함수들은 출판물수를 얻고 표시한다.

3. 계승과 도형

실례 6-3에서 원도형을 표시하는 클라스를 보았다. 물론 원을 내놓고도 다른 도형 즉 직4각형, 3각형과 같은것도 있다. 《도형의 종류》라는 용어는 《도형》이라고 부르는것과 원과 직4각형과 같은 도형의 주어진 종류들사이의 계승관계를 암시한다. 이 관계를 사용하여 서로 련관되지 않은 각이한 도형들을 취급할 때보다 더 견고하고 간단한 프로그람을 만들수 있다.

세개의 파생클라스 Circle, Rect, Tria의 기초클라스인 Shape클라스를 만든다. 여기에 실례 9-6이 있다.

(실례 9-6) 원, 직4각형, 3각형

```
#include <iostream>
using namespace std;
class Shape
{
protected:
   int xCo, yCo;
public:
```

```
    Shape() : xCo(0), yCo(0) {}
    Shape(int x, int y) : xCo(x), yCo(y) {}
    void ShowArea() const { cout << "면적="; }
};
class Circle : public Shape
{
private:
    int radius; // (xCo, yCo)는 중심
public:
    Circle() : Shape(), radius(0) {}
    Circle(int x, int y, int r) : Shape(x, y), radius(r) {}
    void ShowArea() const
    {
        Shape::ShowArea();
        cout << 3.141592 * radius * radius << endl;
    }
};
class Rect : public Shape
{
private:
    int width, height;
public:
    Rect() : Shape(), height(0), width(0) {}
    Rect(int x, int y, int h, int w) : Shape(x, y), height(h), width(w) {}
    void ShowArea() const
    {
        Shape::ShowArea();
        cout << height * width << endl;
    }
};
class Tria : public Shape
{
private:
    int width, height;
public:
    Tria() : Shape(), height(0), width(0) {}
    Tria(int x, int y, int h, int w) : Shape(x, y), height(h), width(w) {}
    void ShowArea() const
    {
        Shape::ShowArea();
        cout << height * width / 2.0 << endl;
    }
};
int main()
{
    Circle cir(40, 12, 5);
    Rect rec(12, 7, 10, 15);
    Tria tri(60, 7, 12, 8);
```

```
    cir.ShowArea();
    rec.ShowArea();
    tri.ShowArea();
    return 0;
  }
```

실행할 때 프로그람은 세개의 각이한 도형 원, 직4각형, 3각형을 창조한다.

모든 도형에 공통인 속성 즉 위치를 Shape클라스에 보관한다. 개별적인 도형들은 자기의 고유한 속성 즉 원은 반경, 직4각형은 높이와 너비를 더 가진다. Shape의 ShowArea()함수는 모든 도형에 일반적인 조작 즉 면적을 표시한다. Circle, Rect, Tria 클라스에서 재정의된 ShowArea()함수는 자기의 면적을 계산한다.

기초클라스 Shape는 추상클라스의 실례이므로 이 클라스의 객체를 만드는것은 아무런 의미도 없다. 그러면 Shape객체가 어떤 도형을 표시하는가? 이 질문에는 일리가 없다. 개별적인 도형이 그 자체를 표시할수 있다. Shape클라스는 오직 모든 도형에 공통적인 특성과 동작들을 정의하기 위하여 존재한다.

제 4 절. 공개와 비공개계승

C++는 클라스성원호출을 조절하는 방법을 제공한다. 그러한 호출조종기구의 하나가 파생클라스들을 선언하는 방법이다. 지금까지의 실례들에서는 공개파생클라스만 사용하였다. 실례로 실례 9-5에서

```
class Manager : public Employee
```

이 명령문에서 public예약어의 효과는 무엇인가?

또한 private의 효과는 무엇인가?

public예약어는 파생클라스의 객체들이 기초클라스의 공개성원함수를 호출할수 있다는것을 지정한다. private를 사용할 때 파생클라스의 객체들은 기초클라스의 공개성원함수들을 호출할수 없으므로 파생클라스의 객체들에서 호출할수 있는 기초클라스의 성원은 하나도 없다.

1. 호출결합

호출가능성이 너무 많으므로 어느것이 동작하고 어느것이 동작하지 않는가를 보여주는 실례를 고찰하는것이 좋다. 여기에 실례 9-7이 있다.

(실례 9-7) 공개와 비공개로 파생된 클라스

```
#include <iostream>
using namespace std;
class A
{
private:
   int privDataA;
```

```
protected:
   int protDataA;
public:
   int pubDataA;
};
class B : public A
{
public:
   void Funct()
   {
      int a;
      a = privDataA; // 오유: 호출불가능
      a = protDataA;
      a = pubDataA;
   }
};
class C : private A
{
public:
   void Funct()
   {
      int a;
      a = privDataA; // 오유: 호출불가능
      a = protDataA;
      a = pubDataA;
   }
};
int main()
{
   int a;
   B objB;
   a = objB.privDataA;     // 오유: 호출불가능
   a = objB.protDataA;     // 오유: 호출불가능
   a = objB.pubDataA;
   C objC;
   a = objC.privDataA;// 오유: 호출불가능
   a = objC.protDataA;// 오유: 호출불가능
   a = objC.pubDataA; // 오유: 호출불가능
   return 0;
}
```

이 프로그람은 비공개, 보호, 공개자료성원들을 가지는 기초클라스 A와 그로부터 파생된 두개의 클라스 B, C를 지정한다. B는 공개파생되고 C는 비공개파생된다.

이미 알고있는것처럼 파생클라스의 함수들은 기초클라스의 보호와 공개자료를 호출할수 있다. 또한 파생클라스의 객체는 기초클라스의 비공개 혹은 보호성원을 호출할 수 없다.

새로운것은 공개파생된 클라스와 비공개파생된 클라스사이의 차이이다. 공개파생

된 클라스 B의 객체는 기초클라스 A의 공개성원들을 호출할수 있으나 비공개파생된 클라스 C의 객체는 기초클라스 A의 공개성원들을 호출할수 없다. 즉 C의 객체는 오직 자기 파생클라스의 공개성원만 호출할수 있다. 이것을 그림 9-6에 주었다.

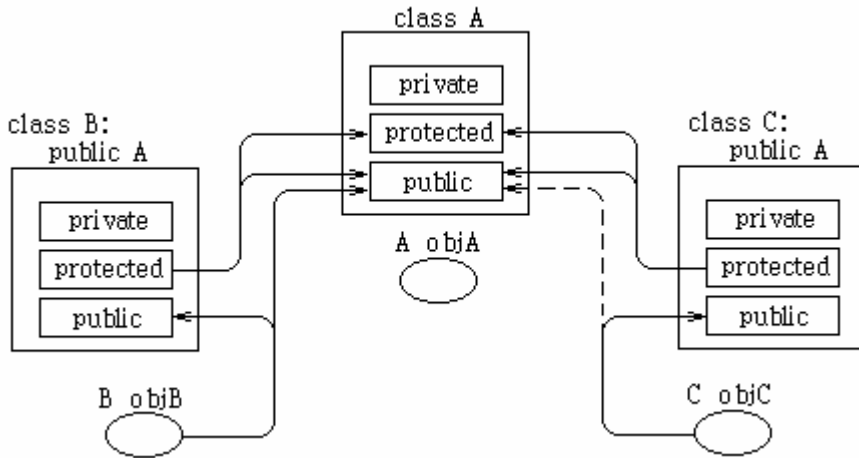



그림 9-6. 공개와 비공개파생

클라스를 창조할 때 호출지정자를 지정하지 않으면 private로 가정된다.

2. 호출지정자의 사용

그러면 공개계승과 비공개계승을 언제 사용하는가?

대부분의 경우에 파생클라스는 기초클라스의 개량판 혹은 전문판을 제공하기 위하여 존재한다. 그러한 파생클라스의 실례를 이미 보았다. 실례로 CountDn클라스는 Counter클라스에 감소연산자를 추가하고 Manager클라스는 Employee클라스의 더 전문화된 판이다. 이러한 경우에 파생클라스의 객체들이 기초조작을 수행하려고 한다면 기초클라스의 공개함수들을 호출하게 하고 파생클라스가 제공하는 전문화된 조작을 수행하려고 한다면 파생클라스의 함수들을 호출하게 한다. 그러한 경우에 공개파생이 적합하다.

그러나 일부 경우에는 기초클라스의 조작을 완전히 변경하기 위하여 혹은 그 원시적대면부를 은페하거나 위장하기 위하여 파생클라스를 창조한다. 실례로 배렬처럼 동작하지만 배렬경계밖으로 벗어나는 첨수에 대한 보호를 제공하는 아주 세밀한 Array클라스가 이미 있다고 가정하자. 그리고 기본배렬대신에 Array클라스를 탄창클라스의 기초로서 사용하려고 한다. Array로부터 Stack를 파생시킬수 있으나 사용자들이 Stack객체를 배렬처럼, 가령 자료항목을 호출하는데 []연산자를 사용하여 다룰것을 바라지 않는다.

Stack객체는 항상 탄창처럼 Push()와 Pop()를 사용하여 다루어야 한다. 즉 Array클라스를 Stack클라스인것처럼 꾸미려고 한다. 이러한 경우에 비공개계승은 파생된 Stack클라스의 객체로부터 Array클라스의 모든 함수들을 은페하게 한다.

3. 계승의 준위

파생클라스로부터 클라스를 파생시킬수 있다. 여기에 그 사실을 보여주는 자그마한 프로그람이 있다.

```
class A {};
class B : public A {};
class C : public B {};
```

여기서 B는 A로부터 파생되고 C는 B로부터 파생된다. 이러한 과정은 임의의 준위까지 확장할수 있다. 즉 D를 C로부터 파생시킬수 있다.

실례 9-5에 Driver(운전수)라고 부르는 Labour의 특수한 부류를 추가하려고 한다. 여기에 Driver클라스의 객체를 포함하는 실례 9-8의 프로그람이 있다.

Driver가 Labour의 일종이므로 Driver클라스는 그림 9-7과 같이 Labour클라스에서 파생시킨다.

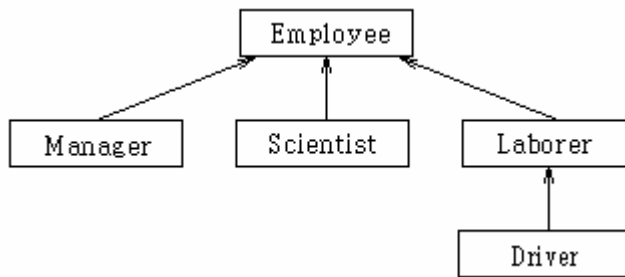



그림 9-7. 실례 9-8의 클라스계층

(실례 9-8) 여러 준위의 계승

```
#include <iostream>
using namespace std;
const int LEN = 80;
class Employee
{
private:
  char name[LEN];
  unsigned long number;
public:
  void GetData()
  {
    cout << "\n  이름? "; cin >> name;
    cout << "\n  종업원 번호? "; cin >> number;
  }
  void PutData() const
  {
    cout << "\n  이름: " << name;
    cout << "\n  종업원 번호: " << number;
  }
```

```
};
class Manager : public Employee
{
private:
   char title[LEN];
public:
   void GetData()
   {
      Employee::GetData();
      cout << "  직위? "; cin >> title;
   }
   void PutData() const
   {
      Employee::PutData();
      cout << "\n  직위: " << title;
   }
};
class Scientist : public Employee
{
private:
   int pubs;
public:
   void GetData()
   {
      Employee::GetData();
      cout << "  출판물 건수? "; cin >> pubs;
   }
   void PutData() const
   {
      Employee::PutData();
      cout << "\n  출판물 건수: " << pubs;
   }
};
class Laborer : public Employee { };
class Driver : public Laborer
{
private:
   int grade;
public:
   void GetData()
   {
      Employee::GetData();
      cout << "  운전급수? "; cin >> grade;
   }
   void PutData() const
   {
      Employee::PutData();
      cout << "\n  운전급수: " << grade;
```

```
    }
};
int main()
{
    Laborer l1;
    Driver d1;
    cout << endl;
    cout << "\n로동자1의 자료를 입력하시오"; l1.GetData();
    cout << "\n운전수1의 자료를 입력하시오"; d1.GetData();
    cout << endl;
    cout << "\n로동자1의 자료"; l1.PutData();
    cout << "\n운전수1의 자료"; d1.PutData();
    cout << endl;
    return 0;
}
```

클라스계층이 조직도표와 같지 않다는것을 기억해두자. 조직도표는 지령행들을 보여준다. 클라스계층은 공통적인 특성을 일반화하는데로부터 생긴다. 클라스가 더 일반적일수록 계층도의 더 높은 준위에 놓인다. Labour는 Driver보다 일반적이고 Driver는 Labour의 특수한 종류이므로 Labour는 클라스계층에서 Driver우에 놓인다.

제 5 절. 다중계승

클라스는 한개이상의 기초클라스로부터 파생될수 있다. 이것을 다중계승(multiple inheritance)이라고 한다. 그림 9-8은 클라스 C를 클라스 A와 D로부터 파생시킬 때 그 고찰방법을 보여준다.

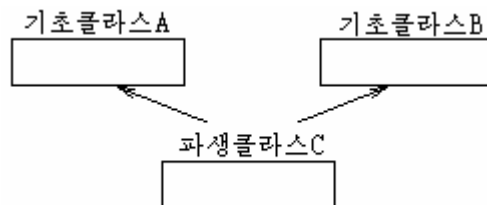



그림 9-8. 다중계승

다중계승의 문법은 단일계승과 같다. 그림 9-8의 관계를 다음과 같이 표시할수 있다.

```
class A {};              // 기초클라스 A
class B {};              // 기초클라스 B
class C: public A, public B {};// C는 A,B로부터 파생된다
```

C의 기초클라스들은 C를 지정할 때 두 점뒤에 반점으로 구분하여 표시한다.

1. 다중계승에서 성원함수

다중계승의 실례로서 실례 9-5에서 일부 종업원들의 학력을 기록한다고 하자. 또

한 다른 프로젝트에서 교육환경을 가지는 대학생을 모의하는 Student라는 클라스를 이미 개발하였다고 하자. 교육자료를 반영하기 위하여 Employee클라스를 변경하지 않고 Student클라스로부터 다중계승에 의해 이 자료를 추가할수 있다.

Student클라스는 마지막으로 졸업한 학교 혹은 대학의 이름과 받은 최종성적을 보관한다. 이 자료항목들은 모두 문자렬로 보관된다. 한개의 성원함수 GetEdu()와 PutEdu()는 사용자에게 이 정보를 묻고 표시한다.

교육정보는 Employee의 모든 클라스와 련관되지 않는다. 교육정보는 관리일군과 과학자에게만 관계되고 Labour에는 기록할 필요가 없다고 가정한다. 그러므로 Manager와 Scientist를 변경하여 그림 9-9와 같이 Employee와 Student클라스로부터 계승하도록 할수 있다.

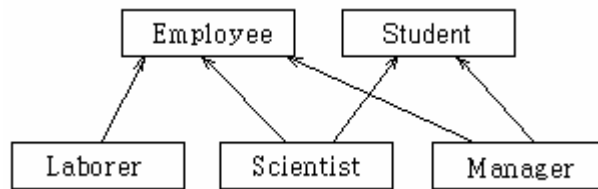



그림 9-9. 실례 9-9에서 클라스계층

여기에 그 관계를 보여주는 간략프로그람이 있다.

```
class Student
     { };
class Employee
     { };
class Manager : private Employee, private Student
     { };
class Scientist : private Employee, private Student
     { };
class Laborer : public Employee
     { };
```

또한 여기에는 구체적인 프로그람이 있다.

(실례 9-9) 다중계승

```
#include <iostream>
using namespace std;
const int LEN = 80;
class Student
{
private:
  char school[LEN];
  char degree[LEN];
public:
  void GetData()
  {
     cout << "\n  대학이름? "; cin >> school;
```

```
        cout << "\n  최종 졸업학교(중학교, 전문학교, 대학)? ";
        cin >> degree;
    }
    void PutData() const
    {
        cout << "\n  대학이름: " << school;
        cout << "\n  최종 졸업학교: " << degree;
    }
};
class Employee
{
private:
    char name[LEN];
    unsigned long number;
public:
    void GetData()
    {
        cout << "\n  이름? "; cin >> name;
        cout << "\n  종업원 번호? "; cin >> number;
    }
    void PutData() const
    {
        cout << "\n  이름: " << name;
        cout << "\n  종업원 번호: " << number;
    }
};
class Manager : private Employee, private Student
{
private:
    char title[LEN];
public:
    void GetData()
    {
        Employee::GetData(); cout << "  직위? "; cin >> title;
        Student::GetData();
    }
    void PutData() const
    {
        Employee::PutData(); cout << "\n  직위: " << title;
        Student::PutData();
    }
};
class Scientist : private Employee, private Student
{
private:
    int pubs;
public:
    void GetData()
```

```
    {
       Employee::GetData(); cout << "  출판물 건수? "; cin >> pubs;
       Student::GetData();
    }
    void PutData() const
    {
       Employee::PutData(); cout << "\n  출판물 건수: " << pubs;
       Student::PutData();
    }
};
class Laborer : public Employee { };
int main()
{
   Manager m1;
   Scientist s1, s2;
   Laborer l1;
   cout << endl;
   cout << "\n관리일군1의 자료를 입력하시오";
   m1.GetData();
   cout << "\n기사1의 자료를 입력하시오";
   s1.GetData();
   cout << "\n기사2의 자료를 입력하시오";
   s2.GetData();
   cout << "\n로동자1의 자료를 입력하시오";
   l1.GetData();
   cout << endl;
   cout << "\n관리일군1의 자료";
   m1.PutData();
   cout << "\n기사1의 자료";
   s1.PutData();
   cout << "\n기사2의 자료";
   s2.PutData();
   cout << "\n로동자1의 자료";
   l1.PutData();
   cout << endl;
   return 0;
}
```

Manager와 Scientist클라스들의 GetData()와 PutData()함수는 Student클라스의 함수호출을 포함한다. 즉

```
Student::GetData();
```

와

```
Student::PutData();
```

Manager와 Scientist클라스가 Student로부터 파생되였으므로 이 함수들을 호출할 수 있다. 여기에 대화가 있다.

```
관리일군1의 자료를 입력하시오
  이름? 김철호
```

```
    종업원 번호? 1
    직위? 지배인
      대학이름? 김책공업종합대학
    최종 졸업학교(중학교, 전문학교, 대학)? 대학
  기사1의 자료를 입력하시오
    이름? 강철
    종업원 번호? 12
    출판물 건수? 20
      대학이름? 김책공업종합대학
    최종 졸업학교(중학교, 전문학교, 대학)? 대학
  기사2의 자료를 입력하시오
    이름? 장향미
    종업원 번호? 22
    출판물 건수? 10
     대학이름? 평양철도대학
    최종 졸업학교(중학교, 전문학교, 대학)? 대학
  로동자1의 자료를 입력하시오
    이름? 리준호
    종업원 번호? 1122
```

실례 9-5와 실례 9-8에서는 정보를 같은 형식으로 표시한다.

실례 9-9에서 Manager와 Scientist클라스는 Employee와 Scientist클라스로부터 비공개로 파생된다. Manager와 Scientist의 객체들이 Employee와 Scientist기초클라스의 함수들을 호출할 필요가 없으므로 공개파생시킬 필요가 없다. 그러나 Labour클라스는 자체의 성원함수를 가지지 않고 Employee의 성원함수에 기초하고있으므로 Employee로부터 공개파생시킨다.

2. 다중계승에서 구성자

실례 9-9에는 구성자가 없다. 구성자를 가지는 실례를 고찰하여 다중계승에서 그것이 어떻게 조종되는가를 고찰하자.

구성자를 포함하는 프로그람을 쓰기로 하자. 이 프로그람이 목재를 제공하는 항목들을 모의한다고 가정하자. 일정한 형의 목재량을 표시하는 클라스를 사용한다. 즉 례를 들면

크기 2×4, 길이 3m짜리 매끈한 판자 100개

클라스는 매개의 목재항목에 대한 각종 자료를 보관하여야 한다.

우선 목재의 길이를 알아야 하고 목재토막의 수와 단가를 보관해야 한다.

또한 목재에 대한 서술을 보관하여야 한다. 이것은 두개의 부분으로 되여있다. 첫부분은 목재자름면의 표준치수들이다. 이것은 센치메터로 주어진다. 실례로 목재의 치수는 5cm×10cm로 주어진다. 이것을 보통 5×10이라고 쓴다. 또한 목재의 질량, 구성물매, 4개걸면 등을 알아야 한다. 이 자료를 보관하는 Type클라스를 창조한다. Type클라스는 표준치수와 목재의 물매용 성원자료를 모두 문자렬로서 즉 "2×6"과 "

건재"와 같이 보관한다. 성원함수는 사용자로부터 이 정보를 얻고 표시한다.

또한 이전 실례의 Distance클라스를 사용하여 길이를 보관한다.

끝으로 Type와 Distance클라스를 둘다 계승하는 Lumber클라스를 창조한다. 여기에 실례 9-10이 있다.

(실례 9-10) 다중계승

```
#include <iostream>
#include <string>
using namespace std;
class Type
{
private:
  string dimensions;
  string grade;
public:
  Type() : dimensions("무효"), grade("무효") {}
  Type(string di, string gr) : dimensions(di), grade(gr) {}
  void GetType()
  {
     cout << "형태상 치수(2x4 등)를 입력하시오: ";
     cin >> dimensions;
     cout << "결면의 등급(rough(거칠다), const(매끈하다), 등을 입력하시오: ";
     cin >> grade;
  }
  void ShowType() const
  {
     cout << "\n  치수: " << dimensions;
     cout << "\n  결면: " << grade;
  }
};
class Distance
{
private:
  int meters;
  float centies;
public:
  Distance () : meters(0), centies(0) { }
  Distance(int me, float ce) : meters(me), centies(ce)  { }
  void GetDist()
  {
     cout << "\n메터를 입력하십시오:"; cin >> meters;
     cout << "센치메터를 입력하십시오:"; cin >> centies;
  }
  void ShowDist() const { cout << meters << "m " << centies << "cm"; }
};
class Lumber : public Type, public Distance // 널판자
{
```

```
private:
   int quantity;
   double price;
public:
   Lumber() : Type(), Distance(), quantity(0), price(0.0) {}
   Lumber(string di, string gr, int me, float ce, int qu, float prc)
      : Type(di, gr), Distance(me, ce), quantity(qu), price(prc) {}
   void GetLumber()
   {
      Type::GetType();
      Distance::GetDist();
      cout << "수량을 입력하시오: "; cin >> quantity;
      cout << "총 가격을 입력하시오: "; cin >> price;
   }
   void ShowLumber() const
   {
      Type::ShowType();
      cout << "\n 판자 " << quantity << "장 "
           << " 한개당 가격 " << price / quantity << "원";
   }
};
int main()
{
   Lumber siding;
   cout << "\n벽널판자의 자료: \n";
   siding.GetLumber();
   Lumber studs("2x4", "const", 8, 0.0, 200, 4.45F);
   cout << "\n벽널판자";
   siding.ShowLumber();
   cout << "\n바닥널판자";
   studs.ShowLumber();
   cout << endl;
   return 0;
}
```

이 프로그람에서 주요한 새 특성은 파생클라스 Lumber에서 구성자의 사용이다. Lumber의 구성자들은 Type와 Distance의 적당한 구성자를 호출한다.

1) 인수없는 구성자

Type에서 인수없는 구성자는

```
Type() : dimensions("무효"), grade("무효") {}
```

이 구성자는 dimensions와 grade변수들에 "무효"(not available)라고 써넣는다. 그리하여 사용자가 초기화되지 않은 Lumber객체의 자료를 표시하려고 시도할 때 주의를 주게 한다. 이미 Distance클라스에서 인수없는 구성자를 추가하였다.

```
Distance () : meters(0), centies(0) { }
```

Lumber의 인수없는 구성자는 이 구성자들을 모두 호출한다.

```
Lumber() : Type(), Distance(), quantity(0), price(0.0) {}
```

두점뒤에 기초클라스구성자들의 이름들을 반점으로 구분하여 배렬한다. Lumber() 구성자를 호출하면 기초클라스구성자 Type()와 Distance()를 실행한다. 또한 quantity와 price속성들도 초기화된다.

2) 여러인수구성자

여기에 Type의 2인수구성자가 있다.

```
Type(string di, string gr) : dimensions(di), grade(gr) {}
```

이 구성자는 문자렬인수들을 dimensions와 grade성원자료항목들에 복사한다.

여기에 Distance의 구성자가 있다.

```
Distance(int me, float ce) : meters(me), centies(ce)  { }
```

Lumber의 구성자는 이상의 구성자들을 모두 호출하므로 그 인수들에 값을 주어야 한다. 또한 추가적으로 자기의 인수가 두개 즉 Lumber의 량과 단가가 있다. 그리하여 Lumber의 구성자는 6개의 인수를 가진다. 이것은 두개의 구성자를 각각 두개 인수씩 사용하여 두번 호출하고 자체의 두개 자료항목을 초기화한다. 여기에 그 수법을 주었다.

```
Lumber(string di, string gr,          // Type용 인수
      int me, float ce,               // Distance용 인수
      int qu, float prc)     :        // 자료
         Type(di, gr),                // Type구성자 호출
         Distance(me, ce),            // Distance구성자 호출
         quantity(qu), price(prc)     // 자료 초기화
   {}
```

3. 다중계승에서 모호성

다중계승을 포함하는 경우에 이상한 문제들이 나타날수 있다. 여기에 보편적인 실례가 있다. 두개의 기초클라스가 같은 이름의 함수를 가지고 두개의 기초클라스로부터 파생된 하나의 클라스에는 이러한 이름을 가지는 함수가 없다.
그러면 파생클라스의 객체가 기초클라스의 함수를 어떻게 정확히 호출하겠는가?

번역프로그람은 두개의 함수들중 어느것을 호출해야 할지 분간할수 없으므로 함수이름만 가지고는 불충분하다. 여기에 그러한 경우를 보여주는 실례가 있다.

(실례 9-11) 다중계승에서 모호성

```
#include <iostream>
using namespace std;
class A
{
public:
  void Show() { cout << "class A\n"; }
};
class B
{
public:
```

```
    void Show() { cout << "class B\n"; }
};
class C : public A, public B { };

int main()
{
    C objC;
    objC.Show(); // 모호하다.
    objC.A::Show();
    objC.B::Show();
    return 0;
}
```

문제는 범위해결연산자에 의하여 함수가 정의되는 클라스를 지정하여 해결할수 있다. 이리하여

```
objC.A::Show();
```

는 A클라스안에 있는 Show()를 참고하고

```
objC.B::Show();
```

는 B클라스안에 있는 Show()를 참고한다. 이것을 모호성의 해결이라고 한다.

다른 종류의 모호성은 어떤 클라스를 같은 클라스로부터 파생된 두개의 클라스로부터 파생시킬 때 제기된다. 이것은 릉형 계승나무를 만든다. 실례 9-12는 그것을 보여준다.

(실례 9-12) 릉형 다중계승

```
#include <iostream>
using namespace std;
class A
{
public: void Func() {}
};
class B : public A {};
class C : public A {};
class D : public B, public C {};
int main()
{
    D objD;
    objD.Func(); // 모호하다
    return 0;
}
```

클라스 B와 C는 모두 A로부터 파생되고 클라스 D는 B와 C로부터 다중계승에 의해 파생된다. 클라스 D의 객체로부터 클라스 A의 성원함수를 호출하면 난관이 생긴다. 이 실례에서 objD는 Func()를 호출한다. 그러나 B와 C는 둘다 A로부터 계승된 Func()의 사본을 가지므로 번역프로그람은 어느 사본을 사용할지 결심할수 없으므로 오유를 경고한다.

이 문제와 관련하여 사본을 만드는 몇가지 방법이 있으나 이러한 모호성이 제기되

는 사실은 많은 전문가들에게 다중계승을 피할것을 권고하게 한다. 경험이 적으면 다중계승을 사용하지 말아야 한다.

제 6 절. 클라스를 포함하는 클라스

1. 클라스를 포함하는 클라스로서 용기

계승에서 클라스 B가 클라스 A로부터 파생된다면 B는 A의 일부라고 말할수 있다. 이것은 B가 A의 특성을 모두 가지며 또한 그 부분이기때문이다. 례를 들면 찌르레기가 새의 일종이라는것과 같다. 찌르레기는 모든 새들이 공유하는 특성(날개, 모양새 등)을 가지지만 자기의 고유한 특성(어두운 진주색 깃털)도 가진다. 그러한 리유로 계승은 자주 관계의 종류라고 한다. 관계의 다른 종류 즉 has-a관계 또는 포함관계가 있다. 찌르레기는 꼬리를 가진다. 이것은 찌르레기가 꼬리의 실례를 포함한다는것을 의미한다.

객체지향프로그람작성법에서 has-a관계는 한 객체가 다른 객체에 포함될 때 발생한다. 여기에 클라스 A의 객체를 클라스 B에 포함하는 경우가 있다.

```
class A {};
class B
{
   A objA;     // objA를 클라스 A의 객체로서 정의한다.
};
```

일부 경우에 계승과 포함관계는 류사한 목적에 사용된다. 실례 9-9를 계승에서 포함을 사용하도록 다시 쓸수 있다. 실례 9-9에서 Manager와 Scientist클라스를 Employee로부터 파생시키고 Student클라스는 계승관계를 사용한다. 새로운 실례 9-13에서 Manager와 Scientist클라스들은 Employee와 Student클라스의 실례를 포함한다. (그림 9-10)

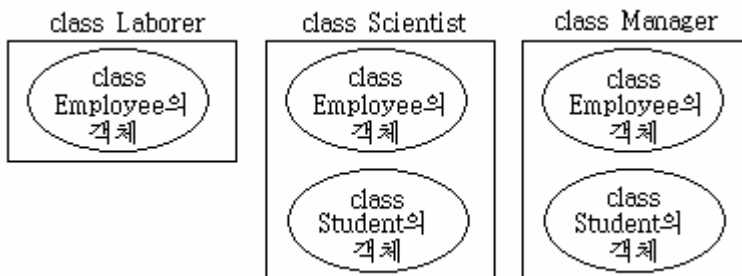



그림 9-10. 실례 9-13에서 클라스계층

다음의 간략프로그람은 다른 방법으로 이 관계를 보여준다.

```
class Student
{ };
```

```
class Employee
{ };
class Manager
{
  Employee emp;
  Student stu;
};
class Scientist
{
  Employee emp;
  Student stu;
};
class Laborer
{
  Employee emp;
};
```

실례 9-13의 원천프로그람은 다음과 같다.

(실례 9-13) 종업원과 학력을 가지는 용기

```
#include <iostream>
#include <string>
using namespace std;
class Student
{
private:
  string school;
  string degree;
public:
  void GetData()
  {
     cout << "\n  대학이름? "; cin >> school;
     cout << "\n  최종 졸업학교(중학교, 전문학교, 대학)? ";
     cin >> degree;
  }
  void PutData() const
  {
     cout << "\n  대학이름: " << school;
     cout << "\n  최종 졸업학교: " << degree;
  }
};
class Employee
{
private:
  string name;
  unsigned long number;
public:
  void GetData()
  {
```

```
        cout << "\n  이름? "; cin >> name;
        cout << "\n  종업원 번호? "; cin >> number;
    }
    void PutData() const
    {
        cout << "\n  이름: " << name;
        cout << "\n  종업원 번호: " << number;
    }
};
class Manager
{
private:
    string title;
    Employee emp;
    Student stu;
public:
    void GetData()
    {
        emp.GetData();
        cout << "  직위? "; cin >> title;
        stu.GetData();
    }
    void PutData() const
    {
        emp.PutData();
        cout << "\n  직위: " << title;
        stu.PutData();
    }
};
class Scientist
{
private:
    int pubs;
    Employee emp;
    Student stu;
public:
    void GetData()
    {
        emp.GetData();
        cout << "  출판물 건수? "; cin >> pubs;
        stu.GetData();
    }
    void PutData() const
    {
        emp.PutData();
        cout << "\n  출판물 건수: " << pubs; stu.PutData();
    }
};
```

```
class Laborer
{
private:
   Employee emp;
public:
   void GetData() { emp.GetData(); }
   void PutData() const { emp.PutData(); }
};
int main()
{
   Manager m1;
   Scientist s1, s2;
   Laborer l1;
   cout << endl;
   cout << "\n관리일군1의 자료를 입력하시오";
   m1.GetData();
   cout << "\n기사1의 자료를 입력하시오";
   s1.GetData();
   cout << "\n기사2의 자료를 입력하시오";
   s2.GetData();
   cout << "\n로동자1의 자료를 입력하시오";
   l1.GetData();
   cout << endl;
   cout << "\n관리일군1의 자료";
   m1.PutData();
   cout << "\n기사1의 자료";
   s1.PutData();
   cout << "\n기사2의 자료";
   s2.PutData();
   cout << "\n로동자1의 자료";
   l1.PutData();
   cout << endl;
   return 0;
}
```

실례 9-13에서도 Student와 Employee클라스는 실례 9-9에서와 같다. 그러나 Manager와 Scientist클라스에 의하여 다른 방법으로 사용된다.

포함은 자료형으로서 사용되는 클라스에서 매우 쓸모있다. 이러한 형의 객체는 int와 같은 기본형처럼 클라스안에서 사용할수 있다.

일반적으로 계승관계는 실현하기 쉽고 명백한 개념적틀거리를 제공한다.

2. 계승과 프로그람개발

프로그람개발과정은 기본적으로 객체지향프로그람작성법에 의해 교체되고있다. 이것은 객체지향프로그람작성법에서 클라스와 계승을 사용하고있기때문이다. 그러면 프로그람개발과정에 대하여 고찰하자.

작성자 A는 클라스를 하나 창조한다. 그것을 Distance클라스라고 하고 사용자정의 자료형에 대하여 동작하는 산수연산을 위한 성원함수들의 한조를 만든다.

작성자 B는 Distance를 개량한 부호있는 거리를 사용하려고 한다. 그러기 위하여 새로운 클라스 즉 실례 9-4와 같이 DistSign을 창조한다. 이때 DistSign은 Distance에서 파생되지만 부호있는 거리를 실현할수 있게 확장한다.

작성자 C와 D는 DistSign클라스를 사용하는 응용프로그람을 쓴다.

이때 작성자 B는 Distance성원함수의 원천코드를 호출할수 없다.

또한 작성자 C와 D는 DistSign의 원천코드를 호출할수 없다.

C++의 쏘프트웨어재리용특성으로 하여 B는 A의 코드를 변경하고 확장할수 있으며 C와 D는 B(와 A)의 코드를 사용할수 있다.

현재 쏘프트웨어도구개발자와 응용프로그람작성자사이의 차이가 거의 없어지고있다. 작성자 A는 일반용프로그람작성도구(Distance클라스)를 창조한다. 작성자 B는 이 클라스의 특정판(DistSign클라스)을 창조한다. 작성자 C와 D는 응용프로그람을 창조한다. A는 도구개발자, C와 D는 응용프로그람개발자이다. B는 그 사이에 있다. 객체지향프로그람작성법은 프로그람작성을 더 융통성있고 복잡하게 한다.

13장에서는 클라스를 의뢰기호출가능부분과 목적형식의 부분으로 갈라서 원천코드를 배포함으로써 다른 프로그람작성자들이 사용할수 있다는것을 설명한다.

요 약

파생클라스는 기초클라스의 특성을 계승한다. 파생클라스는 자기의 고유한 특성을 추가하여 기초클라스의 전용판으로 만들수 있다. 계승은 현존클라스의 능력을 확장하고 계승자체를 사용하여 프로그람을 설계할수 있는 강력한 방도를 준다.

파생클라스로부터, 파생클라스의 객체들로부터 기초클라스성원의 호출능력은 중요한 문제이다. 예약어 protected뒤에 놓이는 기초클라스의 자료와 함수들을 파생클라스로부터 호출할수 있으나 파생클라스의 객체를 비롯한 다른 객체들이 호출할수 없다. 클라스들은 기초클라스로부터 공개 혹은 비공개적으로 파생될수 있다. 공개파생된 클라스의 객체는 기초클라스의 공개성원을 호출할수 있으며 비공개파생된 클라스의 객체는 기초클라스의 공개성원을 호출할수 없다.

클라스는 하나이상의 기초클라스로부터 파생될수 있다. 이것을 다중계승이라고 한다. 또한 클라스를 다른 클라스에 포함할수 있다.

계승은 쏘프트웨어의 재리용성을 가능하게 한다. 파생클라스는 기초클라스를 변경하지 않고도(지어는 기초클라스의 원천코드를 호출하지 않고도) 기초클라스의 능력을 확장할수 있다. 이것은 쏘프트웨어개발자들에게 새로운 융통성을 제공하고 그들의 역할을 더욱 더 높이게 한다.

문 제

1. 계승은

① 일반클라스들을 더 특정한 클라스로 만드는 수단이다.

② 클라스의 객체에 인수를 넘기는 수단이다.

③ 현존클라스에 다시 코드를 쓰지 않고 특성들을 추가하는 수단이다.

④ 자료은페와 밀봉성을 개선하는 수단이다.

어느것이 옳은가?

2. 파생클라스는 기초클라스로부터 파생된다고 말할수 있는가?

3. 계승의 우점으로서는

① 자연선택을 통해 클라스의 확장을 제공하는것

② 클라스서고의 창조를 촉진하는것

③ 코드의 반복쓰기를 없애는것

④ 유용한 개념적 틀거리를 제공한다는것이 포함된다.

어느것이 옳은가?

4. 클라스 Person으로부터 공개적으로 파생되는 클라스 Student의 지정자의 첫행을 쓰시오.

5. 기초클라스에 파생클라스의 추가는 기초클라스에 대한 기본적변경을 요구하는가?

6. 파생클라스의 성원함수로부터 호출하기 위해서는 기초클라스의 자료나 함수가 어떤 호출로 되여야 하는가?

7. 기초클라스가 성원함수 BaseFunc()를 포함하고 파생클라스가 이런 이름을 가진 함수를 포함하지 않는다면 파생클라스의 객체가 BaseFunc()를 호출할수 있는가?

8. 문제 4에서 언급한 클라스를 가정하고 클라스 Person이 PerFunc()라는 성원함수를 포함하고있다. 클라스 Student의 객체 stuObj가 PerFunc()를 호출하는 명령문을 쓰시오.

9. 파생클라스에 구성자를 지정하지 않으면 파생클라스의 객체들은 기초클라스의 구성자를 사용한다. 옳은가?

10. 기초클라스와 파생클라스가 각각 같은 이름의 성원함수를 포함한다면 파생클라스의 객체가 성원함수를 호출할 때 어느것이 호출되는가? 범위해결연산자를 사용하지 않는다고 가정하시오.

11. 문제 4에서 기초클라스 Person의 기정구성자를 호출하는 파생클라스 Student의 인수없는 구성자를 위한 선언자를 쓰시오.

12. 범위해결연산자는 보통

① 변수들의 보임성을 일정한 함수로 제한한다.

② 클라스가 어느 기초클라스로부터 파생되였는가를 말한다.

③ 특정클라스를 지정한다.

④ 모호성을 해결한다. 어느것이 옳은가?

13. 어떤 객체도 창조하지 않는 클라스를 지정하는것이 필요한가?

14. 기초클라스 Base로부터 파생되는 클라스 Derv가 있다고 하자. 1인수를 가지며 기초클라스의 구성자에 따라 이 인수를 넘기는 파생클라스구성자용 선언자를 쓰시오.

15. 클라스 Base로부터 클라스 Derv가 비공개로 파생된다고 하자. main()에 배치된 클라스 Derv의 객체은

① Derv의 공개성원

② Derv의 보호성원

③ Derv의 비공개성원

④ Base의 공개성원

⑤ Base의 보호성원

⑥ Base의 비공개성원을 호출할수 있다.

어느것이 옳은가?

16. 클라스 D는 클라스 C로부터, 클라스 C는 클라스 B로부터, 클라스 B는 클라스 A로부터 파생할수 있는가?

17. 클라스계층은

① 조직도표와 같은 관계를 보여준다.

② has-a관계를 서술한다.

③ is-kind-of관계를 서술한다.

④ 가계와 같은 관계를 보여준다.

어느것이 옳은가?

18. 클라스 Wheel과 클라스 Rubber로부터 파생되는 클라스 Tire용 지정자의 첫 행을 쓰시오.

19. 기초클라스 Base로부터 파생된 클라스 Derv를 가정하시오. 두 클라스는 인수가 없는 성원함수 Func()를 가진다. Derv의 성원함수에서 기초클라스의 Func()를 호출하는 명령문을 쓰시오.

20. 한 클라스의 객체들을 다른 클라스의 성원으로 할수 있는가?

련습문제

1. 도서와 음성카세트를 출판하는 출판사를 고찰하자. 출판물의 제목(string)과 가격(float형)을 보관하는 클라스 Publication을 창조하시오. 이 클라스로부터 두개의 클

라스를 파생하시오. 하나는 Book로서 거기에 페지수(int)를 추가하고 다른 하나는 Tape로서 연주시간(분, int)을 추가하시오. 매개 클라스는 각각 건반을 통하여 사용자로부터 자료를 얻는 GetData()함수와 자료를 표시하는 PutData()함수를 가진다. main()프로그람에서 이 클라스들의 실례를 창조하여 Book와 Tape클라스들을 시험하시오. 즉 GetData()로 자료를 사용자로부터 받아들이고 PutData()로 자료를 표시하시오.

2. 실례 8-14를 고찰하시오. 이 실례의 String클라스는 결함이 있다. 즉 이 클라스의 객체들이 너무 많은 문자들을 가지도록 초기화되는가를 검사하지 못한다.(SZ상수는 값 80을 가진다) 실례로

```
String s = "This string will surely exceed the width of the "
        "source, which is what the SZ constant represents. "
```

기초클라스로부터 문자렬을 가지는 클라스 PString을 파생하시오. PString은 정의에서 너무 긴 문자렬상수가 사용될 때 완충기넘침을 막는다. 파생클라스의 새 구성자는 문자렬상수가 더 길면 str에 SZ개의 문자들만 복사하고 길이가 짧으면 전체 상수를 복사해야 한다. main()프로그람에서 각이한 길이를 가지는 문자렬들을 시험하시오.

3. 련습 1의 Publication, Book, Tape클라스들을 사용하시오. 최근 3달동안 특정한 출판물의 판매액수를 기록할수 있도록 세개의 float배렬을 유지하는 기초클라스 Sales를 추가하시오. 사용자로부터 세개의 판매량을 얻는 함수 GetData()와 판매량을 표시하는 함수 PutData()를 추가하시오. Book와 Tape클라스는 두개의 Publication과 Sales클라스로부터 파생되도록 변경하시오. 클라스 Book 또는 Tape의 객체는 자기 자료에 판매자료까지 입출력해야 한다. main()함수에서 Book객체와 Tape객체를 창조하고 그 입출력을 시험하시오.

4. 련습 1과 3의 출판사에서 도서를 배포하기 위한 셋째 매체 즉 콤퓨터디스크를 추가하시오. Book와 Tape처럼 Publication으로부터 파생되는 Disk클라스를 추가하시오. Disk클라스는 다른 클라스들과 같은 성원함수들을 포함해야 한다. 이 클라스에 유일한 자료항목은 디스크크기 즉 3.5in 혹은 5.25in이다. 이 항목의 보관에 enum Boolean형을 사용하시오. 그러나 전체크기를 표시해야 한다. 사용자는 3 또는 5를 입력하여 적당한 크기를 선택할수 있다.

5. 실례 9-5의 Employee클라스에서 Employee2라는 클라스를 파생시키시오. 또한 Compensation이라는 새 클라스에는 double자료항목을 추가해야 하며 또한 종업원이 시간지불, 주간지불, 월지불을 받는가를 가리키는 Period라는 렬거형을 추가해야 한다. 단순히 Manager, Scientist, Labour클라스들을 변경하여 그것들을 Employee대신에 Employee2로부터 파생시킬수도 있다. 그러나 많은 상태에서 Compensation이라는 기초클라스와 세개의 새 클라스 Manager2, Scientist2, Labour2을 창조하고 원시의 Manager, Scientist, Labour클라스들과 Compensation클라스로부터 세개의 클라스

를 파생시키는 다중계승을 사용하는것이 객체지향프로그람작성에 적합하다. 이 방법은 원시클라스를 변경하지 않는다.

6. 실례 8-12와 같이 SafeArray클라스를 계승을 사용하여 사용자가 구성자에 그 배렬의 상한과 하한을 지정할수 있게 파생시키시오. 이것은 8장의 련습 9와 비슷하다. 다만 새 클라스창조에는 원시클라스를 변경할 대신에 계승을 사용해야 한다.

7. 실례 9-2에서는 앞붙이표기를 사용하여 계수기를 증가 또는 감소할수 있다. 계승을 사용하여 증가와 감소에 뒤붙이표기를 사용할수 있게 기능을 추가하시오.

8. BASIC와 같은 언어에서 연산자는 현존문자렬의 부분을 선택하고 다른 문자렬에 그것들을 대입하게 한다.(표준 C++ string클라스는 다른 방법을 제공한다.) 계승을 사용하여 련습 2의 PString클라스에 그 기능을 추가하시오. 파생클라스 PString2에서는 세개의 새 함수 Left(), Mid(), Right()를 추가하시오.

```
s2.Left(s1, n);        // s2은 s1로부터 제일 왼쪽의 n개 문자렬
s2.Mid(s1, s, n);      // s2은 s1로부터 s번째 문자로부터 n개의 문자렬
s2.Right(s1, n);       // s2은 s1로부터 제일 오른쪽의 n개 문자렬
```

for순환을 사용하여 s1의 적당한 부분을 한문자씩 PString2림시객체에 복사하고 그것을 돌려주시오. 이 함수들은 참고에 의한 귀환을 가지므로 같기기호의 왼변에서 사용하여 현존문자렬의 부분을 변경할수 있다.

9. 련습 1의 Publication, Book, Tape클라스를 사용하시오. Publication클라스로부터 새 클라스 Publication2를 파생하고 여기에 도서와 테프의 출판날자를 포함하시오. 그다음 Book와 Tape를 Publication2로부터 파생하도록 변경하시오. 필요한 성원함수들을 변경하시오. 사용자가 다른 자료와 함께 날자를 입출력하시오. 날자에 대하여 6장 련습 5로부터 Date클라스를 사용하시오. 이 클라스는 세개의 int형 년, 월, 일을 가진다.

10. 공장, 기업소의 관리일군에는 과장이라는 부류가 있다. 실례 9-9의 Manage로부터 Excutive라는 새 클라스를 파생시키시오. Excutive클라스에 자기 직종에서의 근무년한을 추가하시오. 적당한 성원함수들을 추가하여 다른 관리일군자료와 함께 이 자료항목들을 입출력하도록 하시오.

11. 일부 경우에 수들의 쌍을 한단위로 취급해야 한다. 실례로 화면자리표는 x(수평요소)와 y(수직요소)를 가진다. 이러한 수들의 쌍을 표시하는 구조체 Pair를 정의하고 여기에 int성원변수 두개를 추가하시오. 탄창에 Pair변수를 보관하려고 한다. 즉 Push()함수를 호출하여 탄창에 수쌍을 밀어넣고 Pop()함수를 호출하여 수쌍을 얻으려고 한다. Push()의 인수로서 Pair형의 구조체변수를 가지며 Pop()는 돌림값으로써 Pair구조체를 돌려준다. 실례 9-3의 Stack2클라스를 사용하여 그로부터 새 클라스 PairStack를 파생시키시오. 이 새 클라스는 오직 두개의 성원 즉 재정의된 Push()와 Pop()함수를 가진다. PairStack::Push()함수는 쌍으로 된 두개의 옹근수를 보관하기 위하여 Stack2::Push()를 두번 호출해야 하며 PairStack::Pop()함수도 역시

Stack2::Pop()를 두번 호출해야 한다.

12. 8장 련습 7의 분수클라스 Fraction으로부터 데림분수를 모의하는 클라스 Fraction2를 파생시키시오. 데림분수 $2\frac{3}{5}$을 2:3/5형식으로 입력하고 표시하시오. 분수클라스 Fraction에서 재정의한 성원함수들을 재정의하고 main()에서 시험하시오.

제 10 장. 지적자

지적자는 C++프로그람작성에서 제일 어려운 부분의 하나이다. 이 장에서는 지적자를 리해하고 C++프로그람작성에서 실제로 사용해본다.

일반적으로 지적자는 다음과 같은 용도에 사용된다.

· 배렬원소의 호출

· 함수에서 원시인수를 변경해야 할 때 함수에로의 인수넘기기

· 함수에 배렬과 문자렬넘기기

· 체계로부터 기억기의 얻기

· 련결목록과 같은 자료구조체의 창조

지적자는 다른 언어들에서보다 C++에서 더 많이 쓰인다.

그러면 실제로 지적자를 강조할 필요가 있는가?

물론 지적자없이도 많은 일을 할수 있다.

C++에서 지적자를 사용하는 일부 조작은 다른 방법으로 수행할수 있다. 실례로 배렬원소는 지적자표기가 아니라 배렬표기에 의하여 호출할수 있으며 지적자에 의해 함수에 넘긴 인수는 물론 참고에 의해 넘긴 인수들도 변경할수 있다.

그러나 일부 상황에서 지적자는 C++의 능력을 확장해주는 중요한 도구를 제공해준다. 그러한 실례는 련결목록과 2진나무와 같은 자료구조체의 창조이다.

사실상 C++의 일부 중요 특성(즉 가상함수, new연산자, this지적자 등)은 지적자의 사용을 전제로 한다. 그러므로 지적자를 사용하지 않고도 C++프로그람의 대부분을 수행할수 있지만 언어의 대부분의 기능을 얻는데서 지적자는 본질적인 요소로 된다.

이 장에서는 주소상수와 지적자변수의 개념, 배렬과 함수인수, 문자렬에서 지적자의 사용, new와 delete에 의한 기억관리, 지적자와 객체, 련결목록에 대하여 설명한다.

제 1 절. 주소와 지적자

콤퓨터기억기의 매개 바이트는 주소를 가진다. 주소는 거리의 주택에 할당된것과 같은 수값이다. 주소는 0으로 시작하며 1,2,3,…으로 된다. 기억기 1MB를 가지고있다면 가장 큰 주소는 1,048,575이다.

사용자의 프로그람은 기억기에 적재될 때 이 주소의 일정한 범위를 차지한다. 이것은 프로그람안의 매개 변수와 함수가 특별한 주소로 시작된다는것을 의미한다.

그림 10-1은 이것을 보여준다.

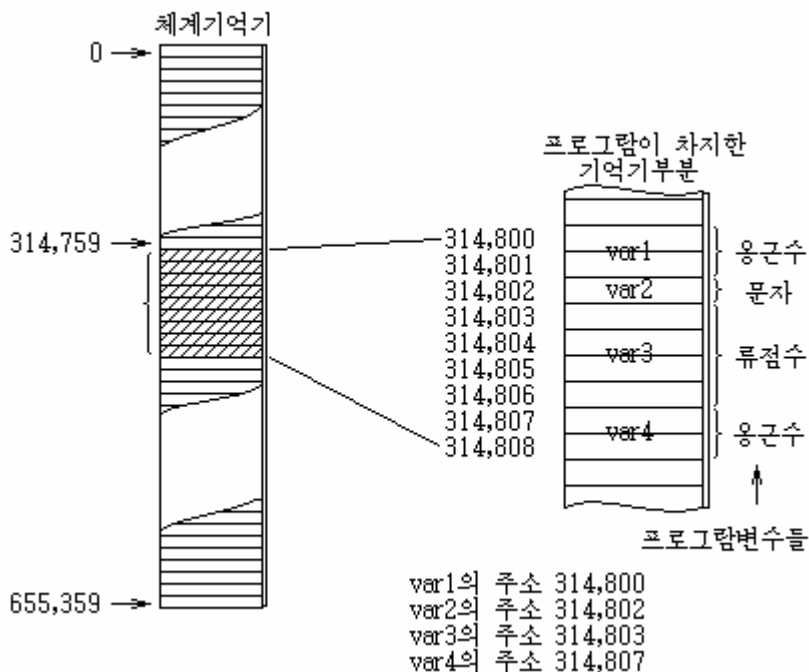



그림 10-1. 기억주소

1. 주소연산자

주소연산자 &를 사용하여 변수가 차지한 주소를 얻을수 있다. 여기에 실례 10-1이 있다.

(실례 10-1) 변수의 주소

```
#include <iostream>
using namespace std;
int main()
{
  int var1 = 11;
  int var2 = 22;
  int var3 = 33;
  cout << &var1 << endl
       << &var2 << endl
       << &var3 << endl;
  return 0;
}
```

실례 10-1은 세개의 옹근수변수를 정의하고 그것들을 11, 22, 33으로 초기화하며 변수들의 주소를 출력한다.

프로그람에서 변수가 차지한 실제주소는 프로그람을 실행하고있는 콤퓨터, 조작체계, 현재 다른 프로그람이 기억기에 있는가 등 많은 인자들에 의존한다.

그러므로 프로그람을 실행할 때마다 같은 주소를 얻을수 없다. 실행결과는 다음과 같다.

0x8f4ffff4 ← var1의 주소
0x8f4ffff2 ← var2의 주소
0x8f4ffff0 ← var3의 주소

변수의 주소는 변수의 내용과는 전혀 다르다. 세개 변수의 내용은 11, 22, 33이다. 그림 10-2는 기억기에서 세개의 변수를 보여준다.

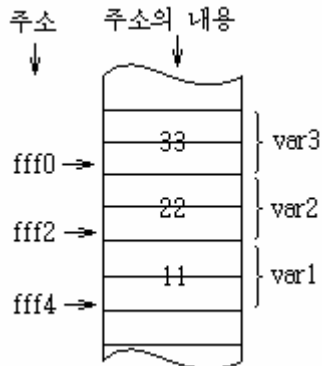



그림 10-2. 주소와 변수의 내용

삽입연산자 <<는 매개 수앞에 앞붙이 0x를 붙여서 주소를 16진수로 해석한다. 이것은 기억기주소를 보여주는 일반적방법이다. 16진수체계를 잘 모른다해도 걱정할 필요는 없다. 실제로 알아야 할것은 매개 변수가 유일한 주소로 시작된다는것이다. 출력으로부터 알수 있는것처럼 매개 변수의 주소는 다음 변수의 주소와 정확히 2byte 차이난다. 이것은 16bit조작체계에서 옹근수가 2byte의 기억기를 차기하기때문이다. char형변수를 사용하면 char가 1byte를 소비하므로 린접주소를 얻을수 있으며 double형을 사용하면 그 주소는 8byte 차이난다.

자동변수가 탄창에 보관되므로 주소는 감소하는 값으로 나타난다. 탄창은 기억기에서 아래로 가면서 커진다. 외부변수를 사용하면 증가하는 주소를 가진다. 그것은 외부변수가 히프에 보관되기때문이다. 히프(heap)는 우로 올라가면서 증가된다. 물론 이렇게 구체적인것까지 알 필요는 없다.

주소연산자 &는 변수를 선언할 때 변수이름의 앞에 붙인다는것과 참고연산자 &는 함수선언이나 정의에서 형이름 뒤에 놓는다는것을 혼돈해서는 안된다.

2. 지적자변수

주소파라메터에는 제한이 있다. 실례 10-1과 같이 객체가 기억기의 어디에 있는가를 찾아내는 방법을 알 필요도 있으나 대체로 주소값출력은 사용하지 않는다.

프로그람작성능력을 높이기 위해서는 추가적인 개념 즉 주소값을 가지는 변수가

요구된다. 이미 문자와 옹근수, 류동소수점수와 같은것을 보관하는 변수형을 보았다.

주소도 역시 보관할수 있다. 주소값을 보관하는 변수를 지적자변수 또는 간단히 지적자(pointer)라고 한다.

그러면 지적자변수의 자료형은 무엇인가?

그것은 주소에 보관되여있는 변수와 같지 않다. 즉 int에로의 지적자는 int형이 아니다.

실례 10-2는 지적자변수의 문법을 보여준다.

(실례 10-2) 지적자(주소변수)

```
#include <iostream>
using namespace std;
int main()
{
  int var1 = 11;
  int var2 = 22;
  cout << &var1 << endl
       << &var2 << endl << endl;
  int* ptr;
  ptr = &var1;
  cout << ptr << endl;
  ptr = &var2;
  cout << ptr << endl;
  return 0;
}
```

이 프로그람은 두개의 옹근수변수 var1과 var2를 정의하고 각각 1과 2로 초기화한 다음 그 주소를 출력한다. 그다음 지적자변수를 정의한다.

```
int * ptr;
```

지적자변수를 정의할 때 별표를 사용한다. 별표는 지적자를 의미한다. 이처럼 명령문은 변수 ptr를 int에로의 지적자로 정의한다. 이것은 그 변수가 옹근수변수의 주소를 보관할수 있다는것을 알려준다.

그러면 임의의 자료형에로의 지적자들을 보관하는 범용지적자형과 무엇이 다른가?

Pointer형을 호출하려면 다음과 같이 선언한다.

```
Pointer ptr;
```

문제는 번역프로그람이 지적자가 가리키는 변수의 종류를 알아야 한다는데 있다.(지적자와 배렬에 대하여 론할 때 그것이 필요하다.) C++에서는 문법적으로 임의의 형에로의 지적자를 선언할수 있다. 실례로

```
char * cptr;
int   * iptr;
float * fptr;
Distance * distptr;
```

3. 문법의 모호성

지적자정의를 쓸 때 형보다도 변수이름앞에 별표를 놓는것이 더 일반적인 방법이다.

```
char *charptr;
```

번역프로그람에는 문제가 없으나 형뒤에 별표를 놓으면 별표가 이름자체의 부분이 아니라 변수형의 부분이라는것을 강조한다. 한행에 같은 형의 지적자를 하나이상 정의할 때 별표를 형으로서 삽입하면 그것을 하나만 삽입하면 될것같지만 실제로는 매개 변수이름앞에 별표를 놓아야 한다.

```
char* ptr1, * ptr2, * ptr3;
```

그러나 이름앞에 별표를 붙여쓰면 다음과 같다.

```
char *ptr1, *ptr2, *ptr3;
```

- 지적자는 값을 가져야 한다.

0x8f4ffff4와 같은 주소는 지적자상수이고 ptr와 같은 지적자는 지적자변수이다. 옹근수변수 var1에 상수값 11을 대입할수 있는것처럼 지적자변수 ptr에도 상수값 0x8f4ffff4를 대입할수 있다.

변수를 정의할 때 초기화하지 않은 경우 변수는 값을 가지지 않는다.

변수가 의미가 없는 오물(garbage)값을 가질수 있다. 지적자의 경우에 오물값은 기억기안의 어떤 주소이지만 그것은 우리가 요구하는 값이 아니다. 그러므로 지적자를 사용하기 전에 지적자에 고유값을 주어야 한다. 실례 10-2에서는 우선

```
ptr = &var1;
```

에 의해 ptr에 var1의 주소를 대입한다.

그다음 ptr에 &var1의 주소를 대입하고 ptr의 값을 출력한다.

그리고 같은 지적자변수 ptr에 var2의 주소를 할당하고 그 값을 출력한다. 그림 10-3은 실례 10-2의 연산을 보여준다. 아래에 그 출력이 있다.

0x8f5ffff4 ← var1의 주소
0x8f5ffff2 ← var2의 주소
0x8f5ffff4 ← ptr는 var1의 주소로 설정된다.
0x8f5ffff2 ← ptr는 var2의 주소로 설정된다.

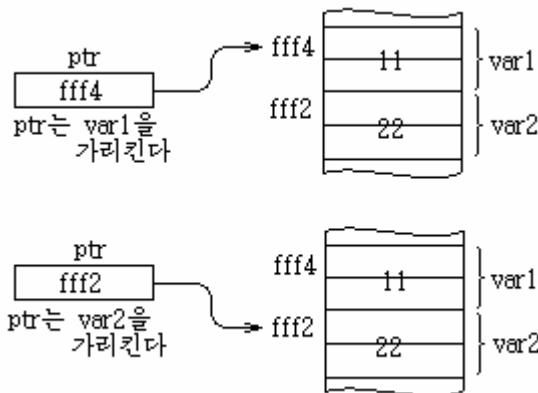



그림 10-3. ptr값의 변경

이와 같이 지적자는 정확한 형의 임의의 변수의 주소를 보관할수 있으므로 주소를 보관하는 그릇이다. 그러나 지적자에는 어떤 값이 주어져야 한다. 그렇지 않으면 프로그람코드 혹은 조작체계와 같은 불필요한 주소를 가리키게 된다. 번역프로그람이 옳지 못한 지적자값에 대하여 오유를 경고하지 않으므로 체계중단이 발생하고 오유수정을 어렵게 한다. 그러므로 반드시 지적자변수를 사용하기 전에 거기에 유효한 주소를 주었는가 확인해야 한다.

4. 지적된 변수의 호출

변수이름을 모르지만 그 주소를 알 때 변수내용을 호출할수 있는가?

변수이름대신 그 주소를 사용하여 변수의 값을 호출하는 문법이 있다. 실례 10-3은 그 방법을 보여준다.

(실례 10-3) 지적자변수호출

```
#include <iostream>
using namespace std;
int main()
{
  int var1 = 11;
  int var2 = 22;
  int* ptr;
  ptr = &var1;
  cout << *ptr << endl;
  ptr = &var2;
  cout << *ptr << endl;
  return 0;
}
```

이 프로그람은 ptr의 주소를 출력하지 않고 ptr에 보관된 주소에 기억되여있는 옹근수값을 출력하는것을 제외하면 실례 10-2와 비슷하다. 출력은 다음과 같다.

```
11
22
```

변수 var1과 var2를 호출하는 식은 *ptr이며 이것은 두개의 cout명령문의 매개에서 나타난다.

*ptr식과 같이 변수이름앞에 별표를 사용하는데 이것을 간접연산자(indirection operator)라고 한다. 식 *ptr는 ptr에 의해 지적된 변수의 값을 표시한다. ptr가 var1의 주소로 설정될 때 var1은 11이므로 식 *ptr는 값 11을 가진다. ptr가 var2의 주소로 변경되면 var2이 22이므로 식 *ptr는 값 22을 얻는다.

간접연산자는 때때로 연산자의 내용을 호출한다. 그림 10-4는 이것을 보여준다.

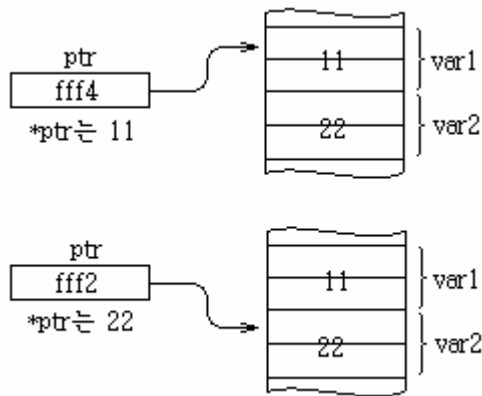



그림 10-4. 지적자를 통한 호출

변수값의 표시뿐아니라 변수에 대하여 직접 수행하는 조작을 처리하는데 지적자를 사용할수 있다. 여기서 변수에 값을 대입하고 그 값을 다른 변수에 대입하는데 지적자를 사용하는 실례 10-4가 있다.

(실례 10-4) 지적자를 사용한 호출

```
#include <iostream>
using namespace std;
int main()
{
   int var1, var2;
   int* ptr;
   ptr = &var1;
   *ptr = 37;
   var2 = *ptr;
   cout << var2 << endl;
   return 0;
}
```

간접연산자로 사용되는 별표는 지적자변수선언에 사용되는 별표와 다른 의미를 가진다. 간접연산자는 변수앞에 놓이며 그것에 의해 지적된 값을 의미한다. 선언에서 사용하는 별표는 지적자를 의미한다.

```
int* ptr;      // int에로의 지적자선언
*ptr = 37;     // ptr에 의해 간접적으로 지적된 변수의 값
```

주소에 보관된 값을 호출하는데 간접연산자를 사용하는것을 지적자간접호출 혹은 비참고(dereference)라고 한다.

다음에 이상을 요약한 실례가 있다.

```
int v;     // int형의 변수 v를 정의한다.
int* p;    // int형의 지적자로서 p를 정의한다.
p = &v;    // 지적자 p에 변수 v의 주소를 대입한다.
v = 3;     // v에 3을 대입한다.
*p = 3;    // 역시 v에 3을 대입한다.
```

마지막 두개의 명령문은 이름에 의해 변수를 참고하는 표준 혹은 직접주소화와 그

주소를 사용하여 변수를 참고하는 지적자 혹은 간접주소화의 차이를 보여준다. 이 장의 실례들에서 변수호출에서 지적자변수사용의 우점이 실제로 지적자에 의해 변수를 직접 호출할수 있게 하는것이라는것을 알수 있다.

지적자는 변수를 직접 호출할수 없을 때 사용된다.

5. void에로의 지적자

지적자의 동작을 보기전에 지적자자료형의 한가지 특성을 알아야 한다. 보통 지적자에 보관한 주소는 그 지적자와 같은 형이여야 한다. int에로의 지적자에 float형변수의 주소를 보관할수 없다. 실례로

```
float floVar = 98.6;
int* ptrInt = &floVar;    // 오유: float를 int에 보관할수 없다.
```

그러나 여기에 례외가 있다. 임의의 자료형을 가리킬수 있는 범용지적자가 있다. 이것을 void에로의 지적자라고 부르며 다음과 같이 정의한다.

```
void* ptr;  // 이때 ptr는 임의의 자료형을 가리킬수 있다.
```

이러한 지적자는 지적된 자료형과는 다르게 조작하는 함수에로의 지적자넘기기와 같은 일정한 용도를 가지고있다.

다음의 실례에서는 void를 사용하지 않는 경우 지적자에 그것과 같은 형의 주소를 대입하도록 하는데 주의해야 한다.

(실례 10-5) void형에로의 지적자

```
#include <iostream>
using namespace std;
int main()
{
    int intVar;
    float floVar;
    int* ptrInt;
    float* ptrFlo;
    void* ptrVoid;
    ptrInt = &intVar;
// ptrInt = &floVar;          // 오유
// ptrFlo = &intVar;          // 오유
    ptrFlo = &floVar;
    ptrVoid = &intVar;
    ptrVoid = &floVar;
    return 0;
}
```

ptrInt에 intVar의 주소를 대입하는것은 그것들이 모두 int*이므로 가능하다. 그러나 ptrInt에 floVar주소를 대입하면 하나는 float*형이고 다른 하나는 int*형이므로 불가능하다. 그러나 ptrVoid는 void에로의 지적자이므로 int*와 같은 임의의 지적자값을 가질수 있다.

만일 어떤 형의 지적자를 다른 형의 지적자에 대입하는데 reinterpret_cast를 사용한다. 실례 10-5에서 설명문이 있는 행들을 다음과 같이 변경한다.

```
ptrInt = reinterpret_cast<int*>(floVar);
ptrFlo = reinterpret_cast<float*>(intVar);
```

물론 이런 방법으로 reinterpret_cast를 사용할것을 권고하지 않지만 이것은 경우에 따라서 복잡한 상황에서 벗어나는 유일한 방법이다. 정적강제형변환은 지적자에 의거하여 작업하지 않는다. C형의 강제형변환을 사용할수도 있지만 C++에서는 좋지 않은 방법이다. (3장 참고)

제 2 절. 지적자와 배렬

지적자와 배렬사이에는 밀접한 련관이 있다. 7장에서 배렬원소를 호출하는 방법을 보았다. 실례 10-6에 그 방법을 보여준다.

(실례 10-6) 배렬표기를 사용하여 호출하는 배렬

```
#include <iostream>
using namespace std;
int main()
{
  int intArray[5] = { 31,54, 77, 52, 93 };
  for(int j=0; j<5; j++)
     cout << intArray[j] << endl;
  return 0;
}
```

cout명령문은 배렬의 매개 원소를 차례로 출력한다. 실례로 j가 3일 때 식 intArray[j]는 값 intArray[3]을 가지므로 배렬의 네번째 원소인 옹근수 52를 호출한다. 여기에 실례 10-6의 출력이 있다.

```
31
54
77
52
93
```

배렬의 원소는 배렬표기는 물론 지적자표기에 의하여서도 호출할수 있다. 실례 10-7에서는 실례 10-6과 달리 지적자표기를 사용한다.

(실례 10-7) 지적자표기법에 의한 배렬의 호출

```
#include <iostream>
using namespace std;
int main()
{
  int intArray[5] = { 31,54, 77, 52, 93 };
  for(int j=0; j<5; j++)
     cout << *(intArray + j) << endl;
```

```
    return 0;
  }
```

실례 10-7에서 식 *(intArray +j)는 실례 10-6의 intArray[j]와 같은 효과를 가지고 프로그람의 출력은 같다.

그러면 식 *(intArray + j)를 어떻게 해석하겠는가?

j가 3이고 식이 *(intArray + 3)이라고 가정하자. 그러면 이것은 배렬의 네번째 원소의 내용을 표시한다.

배렬의 이름은 그 주소이므로 식 intArray+j도 거기에 더해진 어떤 값을 가지는 주소이다. intArray+3이 intArray에 3byte를 더하는것으로 생각할수 있다. intArray는 옹근수배렬이고 이 배렬에 3byte를 더하면 두번째 요소의 가운데로 된다. 그러나 우리는 배렬의 네번째원소를 얻으려고 하지 네번째 바이트를 얻으려고 하지 않는다. (그림 10-5)

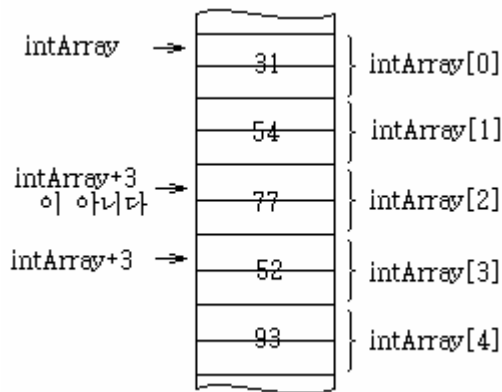



그림 10-5. 옹근수형에 의한 계수

실제로 C++번역프로그람은 자료주소에 대한 산수연산을 할 때 자료의 크기를 고려한다. 번역프로그람은 intArray가 int형의 배렬이라는것을 알고있다. 따라서 식 intArray+3과 만나면 intArray의 네번째 바이트가 아니라 네번째 옹근수의 주소로 해석한다.

그런데 네번째 배렬원소의 주소가 아니라 값을 요구하므로 간접연산자 *를 사용한다. 결과식은 j가 3일 때 *(intArray + 3)으로 되고 그것은 배렬의 네번째 원소의 값 즉 52이다.

그러면 지적자선언이 왜 지적된 변수의 형을 포함해야 하는가에 대하여 고찰하자.

번역프로그람은 지적자가 int에로의 지적자인가, double에로의 지적자인가를 알아야 배렬의 원소들을 호출하여 산수연산을 정확히 처리할수 있다. 번역프로그람은 int형의 경우에 첨수값에 2(16bit조작체계)를 곱하지만 double의 경우에는 8을 곱한다.

- 지적자상수와 지적자변수

배렬주소를 통하여 intArray에서 j를 증가시키지 않고 증가연산자를 사용하려고

한다.

*(intArray++)라고 쓸수 있는가?

그렇게 쓸수 없다. 그것은 상수를 증가시킬수 없기때문이다. 식 intArray는 배렬을 보관하기 위하여 체계가 선택한 주소로서 프로그람이 완료할 때까지 배렬은 이 주소에 상주한다. 즉 intArray는 지적자상수이다. 7++라고 할수 없듯이 intArray++라고 할수 없다.(다중과제처리체계에서 변수주소는 프로그람실행시에 변경될수 있다. 동작하는 프로그람은 디스크로 절환되고 다른 기억위치를 지적하게 된다.)

주소는 증가시킬수 없지만 주소를 보관하는 지적자는 증가시킬수 있다. 다음의 실례 10-8은 그것을 보여준다.

(실례 10-8) 지적자에 의해 호출하는 배렬

```
#include <iostream>
using namespace std;
int main()
{
   int intArray[5] = { 31,54, 77, 52, 93 };
   int* ptrInt;
   ptrInt = intArray;
   for(int j=0; j<5; j++)
      cout << *(ptrInt++) << endl;
   return 0;
}
```

여기서는 int에로의 지적자 ptrInt를 정의하고 거기에 배렬의 주소인 값 intArray를 대입한다. 그러면 다른 식을 사용하여 배렬원소들의 내용을 호출할수 있다.

*(ptrInt++)

변수 ptrInt는 intArray와 같은 주소값으로 시작한다. 즉 값 31을 가지는 첫째 배렬원소 intArray[0]이 처음에 호출된다. 그러나 ptrInt는 상수가 아니라 변수이므로 증가시킬수 있다. ptrInt는 증가된 후 배렬의 둘째 원소 intArray[1]을 가리킨다. 따라서 식 *(ptrInt++)는 배렬의 둘째 원소의 내용 54를 표시한다. 순환은 이 식이 매개 배렬원소를 차례로 호출하게 한다. 실례 10-8의 출력은 실례 10-7과 같다.

제 3 절. 지적자와 함수

5장에서 함수에 인수를 넘기는 세가지 방법 즉 값에 의한 넘기기와 참고에 의한 넘기기, 지적자에 의한 넘기기를 언급하였다.

프로그람에서 함수가 호출하는 변수들을 변경하려면 변수들을 값에 의해 넘기지 말아야 한다. 그것은 함수가 그 변수의 유일한 사본을 얻기때문이다. 이러한 경우에는 참고인수나 지적자를 사용할수 있다.

1. 단순변수넘기기

우선 참고에 의해 인수를 넘기는 방법을 보고 그다음 이것을 지적자인수넘기기와 비교한다. 실례 10-9는 참고에 의한 넘기기를 보여준다.

(실례 10-9) 참고에 의하여 넘기는 인수

```
#include <iostream>
using namespace std;
void Centimize(double&);
int main()
{
   double var = 10.0;
   cout << "var=" << var << "in" << endl;
   Centimize(var);
   cout << "var=" << var << "cm" << endl;
   return 0;
}
void Centimize(double& v)
{
   v *= 2.54;
}
```

여기서는 main()의 변수 var를 인치로부터 센치메터로 변환한다. 함수 Centimize()에 변수를 참고에 의해 넘긴다. (함수선언에서 double자료형뒤의 &는 인수가 참고에 의해 넘어간다는것을 의미한다.) Centimize()함수는 원시변수에 2.54를 곱한다.

그러면 함수가 변수를 참고하는 방법을 고찰하자.

단순히 인수이름 v를 사용하며 v와 var는 같은 객체에 대한 다른 이름이다.

일단 var를 센치메터로 변환하면 main()은 그 결과를 출력한다. 여기에 실례 10-9의 출력이 있다.

```
var=10in
var=25.4cm
```

다음의 실례 10-10은 지적자를 사용할 때 실례 10-9와 등가한 상황을 보여준다.

(실례 10-10) 지적자에 의하여 넘기는 인수

```
#include <iostream>
using namespace std;
void Centimize(double*);
int main()
{
   double var = 10.0;
   cout << "var=" << var << "in" << endl;
   Centimize(&var);
   cout << "var=" << var << "cm" << endl;
   return 0;
}
```

```
void Centimize(double* ptrd)
{
   *ptrd *= 2.54;
}
```

실례 10-10의 출력은 실례 10-9와 같다. 함수 Centimize()는 double에로의 지적자인 인수를 가리키는것으로 선언한다.

```
void Centimize(double *);
```

main()이 함수를 호출할 때 변수의 주소가 인수로서 공급된다.

```
Centimize(&var);
```

이것은 참고에 의해 넘길 때처럼 변수자체가 아니라 변수의 주소이다. Centimize() 함수에 주소를 넘기므로 이 주소에 보관된 값을 호출하려면 간접연산자를 사용해야 한다. 즉

```
*ptrd *= 2.54;
```

물론 이것은 다음과 같다.

```
*ptrd = *ptrd * 2.54;
```

여기서 독립적인 *는 곱하기를 의미한다.

ptrd가 var의 주소를 보관하므로 *ptrd에 대하여 수행하는 조작은 실제로 var에 수행하는것으로 된다. 그림 10-6은 함수안에서 *ptrd의 변경이 호출측프로그람안의 var를 어떻게 변경하는가를 보여준다.

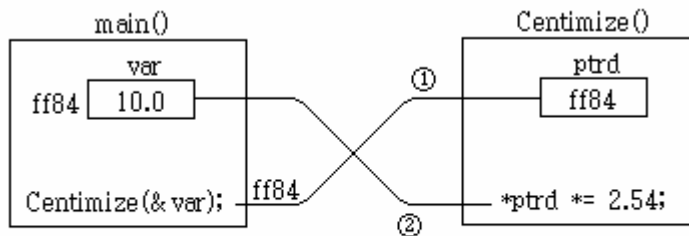



그림 10-6. 함수에 지적자넘기기

함수인수로서 지적자넘기기는 참고에 의한 넘기기와 류사하다. 이것들은 둘다 호출측프로그람안의 변수를 함수에 의해 변경시킬수 있게 한다. 그러나 기구는 다르다. 참고는 원시변수의 별명이고 지적자는 변수의 주소이다.

2. 배렬넘기기

7장으로부터 시작하여 함수에 배렬을 인수로 넘기는 실례들을 몇가지 보았다.

지금까지는 지적자를 배우지 않았으므로 배렬표기를 사용하였다. 그러나 함수에 배렬을 넘기는 방법에는 배렬표기보다 더 일반적인 표기인 지적자표기가 있다. 실례 10-11에서 그것을 보여준다.

(실례 10-11) 지적자에 의한 배렬넘기기

```
#include <iostream>
using namespace std;
const int MAX = 5;
void Centimize(double*);
int main()
{
  double varray[MAX] = { 10.0, 43.1, 95.9, 59.7, 87.3 };
  Centimize(varray);
  for(int j=0; j<MAX; j++)
     cout << "varray[" << j << "]=" << varray[j] << "cm" << endl;
  return 0;
}
void Centimize(double* ptrd)
{
  for(int j=0; j<MAX; j++)
     *ptrd++ *= 2.54;
}
```

함수의 원형선언은 실례 10-10과 같다. 함수의 유일한 변수는 double에로의 지적자이다. 배렬표기에서는 다음과 같이 쓴다.

```
void Centimize(double[]);
```

즉 double*은 double[]과 등가하지만 지적자표기를 더 많이 사용한다.

배렬이름은 배렬의 주소이므로 함수를 호출할 때 주소연산자 &를 사용할 필요가 없다.

```
Centimize(varray); // 배렬의 주소를 넘긴다.
```

Centimize()에서는 배렬주소를 변수 ptrd에 보관하고 배렬의 매개 원소를 차례로 가리키기 위하여 ptrd를 증가시킨다. 즉

```
*ptrd++ *=2.54;
```

그림 10-7은 배렬을 호출하는 방법을 보여준다. 실례 10-11의 출력은 다음과 같다.

```
varray[0]=2.54cm
varray[1]=109.474cm
varray[2]=243.384cm
varray[3]=151.638cm
varray[4]=221.74cm
```

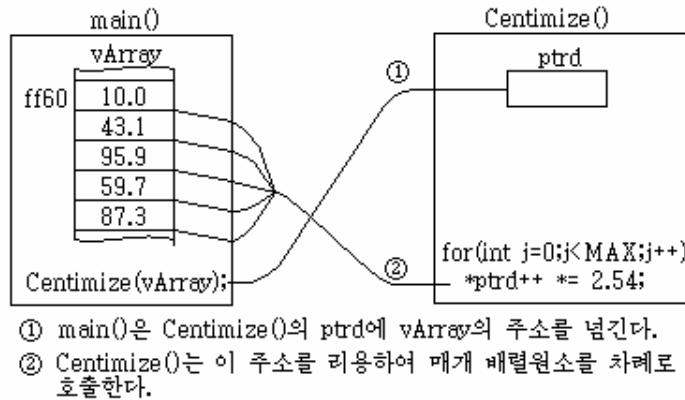



그림 10-7. 함수로부터 배렬호출

여기에 문법적질문이 있다.

번역프로그람이 식 *ptrd++를 *(ptrd++)로 해석하는가, (*ptrd)++로 해석하는가?

간접연산자로 사용하는 *와 ++는 우선순위가 같다. 그러나 같은 우선순위를 가지는 연산자들은 두번째 방법 즉 결합순서(associvity)에 의해 구별된다. 결합순서는 번역프로그람이 오른쪽에 있는 연산수로부터 연산을 시작하는가, 왼쪽에 있는 연산수로부터 연산을 시작하는가 하는것과 관계된다. 한조의 연산자들이 오른쪽결합을 가진다면 번역프로그람은 식의 오른변에 대하여 먼저 연산하고 그다음 왼변에 대하여 연산한다. *, ++와 같은 단항연산자들은 오른쪽 결합을 가진다. 그러므로 식은 *(ptrd++)로 평가되고 그것이 가리키는 값이 아니라 지적자를 증가시킨다. 즉 지적자가 먼저 증가하고 결과로 생기는 주소에 간접연산자가 적용된다.

3. 배렬원소의 분류

배렬원소호출에 지적자를 사용하는 방법으로서 배렬의 내용을 정렬하는 경우를 고찰하자.

두개의 실례를 고찰한다.

1) 지적자를 사용한 정렬

첫째 프로그람은 실례 5-11과 비슷하지만 참고대신 지적자를 사용한다. 인수로서 넘어온 두 수들중에서 둘째 수가 첫째 수보다 작으면 그것들을 교체하는 방법으로 두 수를 정렬한다. 여기에 실례 10-12가 있다.

(실례 10-12) 지적자에 의한 두개 인수의 정렬

```
#include <iostream>
using namespace std;
void Order(int*, int*);
int main()
{
  int n1 = 99, n2 = 11;
  int n3 = 22, n4 = 88;
```

```
    Order(&n1, &n2);
    Order(&n3, &n4);
    cout << "n1=" << n1 << endl;
    cout << "n2=" << n2 << endl;
    cout << "n3=" << n3 << endl;
    cout << "n4=" << n4 << endl;
    return 0;
}
void Order(int* numb1, int* numb2)
{
    if(*numb1 > *numb2)
    {
        int temp = *numb1;
        *numb1 = *numb2;
        *numb2 = temp;
    }
}
```

함수 Order()는 정렬하려는 수들의 주소를 넘기는것을 제외하면 실례 5-11과 같고 지적자에 기초하여 수들을 호출한다. 즉 *numb1은 main()에서 첫 인수로 넘어온 수를 호출하고 *numb2는 둘째 수를 호출한다. 여기에 실례 10-12의 출력이 있다.

```
n1=11
n2=99
n3=22
n4=88
```

옹근수배렬을 정렬하는 다음의 실례 10-13에서 실례 10-12의 Order()함수를 사용한다.

(실례 10-13) 지적자를 사용한 배렬의 정렬

```
#include <iostream>
using namespace std;
int main()
{
    void BSort(int*, int);
    const int N = 10;
    int arr[N] = { 37, 84, 62, 91, 11, 65, 57, 28, 19, 49 };
    BSort(arr, N);
    for(int j=0; j<N; j++)
        cout << arr[j] << " ";
    cout << endl;
    return 0;
}
void BSort(int* ptr, int n)
{
    void Order(int*, int*);
    int j, k;
    for(j=0; j<n-1; j++)
        for(k=j+1; k<n; k++)
```

```
            Order(ptr + j, ptr + k);
    }
    void Order(int* numb1, int* numb2)
    {
      if(*numb1 > *numb2)
      {
         int temp = *numb1;
         *numb1 = *numb2;
         *numb2 = temp;
      }
    }
```

main()에서는 옹근수배렬 arr를 정렬되지 않은 옹근수들로 초기화하고 배렬의 주소와 원소개수를 BSort()함수에 넘긴다. BSort()에서 배렬을 정렬한 다음 main()에서 정렬된 값들을 출력한다. 여기에 실례 10-13의 출력이 있다.

```
11 19 28 37   49 57 62 65 84 91
```

2) 거품분류

BSort()함수는 거품분류(bubble sorting)방법으로 배렬을 정렬한다. 이것은 간단한 정렬방법이다. 배렬안의 수들을 커지는 순서로 배치하려고 한다고 가정하고 그 동작을 고찰하자.

우선 배렬의 첫 원소(arr[0])를 다른 원소들(2번째부터 시작하여)과 차례로 비교한다. 만일 첫 원소가 다른 원소들중 어느 하나보다 크면 두 원소를 서로 교체한다. 이것이 수행되면 적어도 첫째 원소가 제 순서로 놓인다. 즉 제일 작은 원소로 된다. 다음에는 둘째 원소를 셋째 원소부터 시작하여 배렬의 다른 원소들과 차례로 비교하여 더 크면 서로 교체한다. 이렇게 하면 둘째 원소는 두번째로 작은 원소로 된다. 이 과정을 마지막 원소의 앞원소까지의 모든 원소들에 대하여 계속하면 배렬은 정렬된것으로 된다. 그림 10-8은 거품분류의 동작을 보여준다.

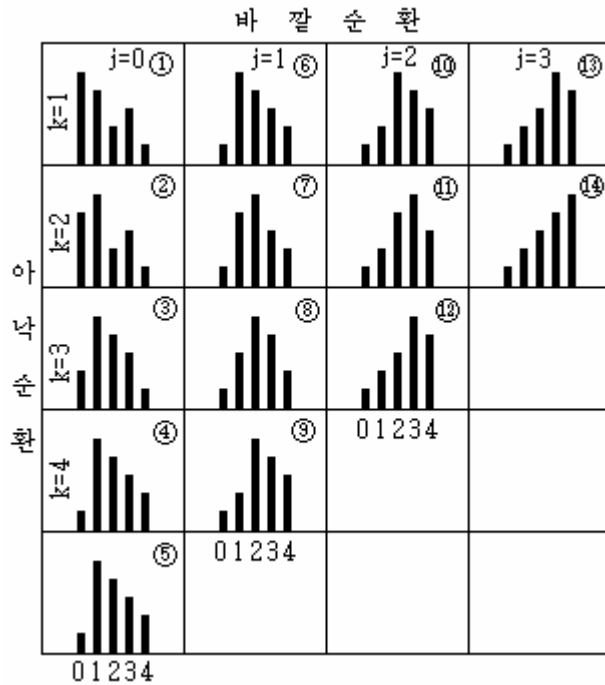



그림 10-8. 거품분류의 동작

실례 10-13은 첫 위치의 수 37을 매개 원소와 차례로 비교하고 11과 교체한다. 다음에 84로 시작하는 둘째 위치의 수를 매개 요소와 비교하여 62와 교체하고 그다음 62를 37과 교체하며 37을 28과 교체하고 28을 19와 교체한다. 셋째위치의 수 84를 다시 62와 교체하고 62를 57과, 57을 37과, 37을 28과 교체한다. 이 과정을 배렬이 정렬될 때까지 계속한다.

실례 10-13의 BSort()함수는 두개의 2중순환으로 이루어진다. 하나는 지적자로 조종한다. 바깥순환은 순환변수 j를 사용하고 안쪽 순환은 순환변수 k를 사용한다. 식 ptr+j와 ptr+k는 순환변수들에 의해 결정되는 배렬의 매개 원소들을 지적한다. 식 ptr+k는 첫 원소(웃끝)로부터 시작하여 마지막 원소(바닥)의 하나 앞까지 한개 옹근수씩 배렬의 아래로 이동한다. 바깥순환에서 ptr+j에 의해 얻어지는 매개 위치에 대하여 ptr+j와 ptr+k가 지적하는 원소들을 Order()함수가 비교하여 첫째가 둘째보다 더 크면 그것들을 교체한다. 그림 10-9는 그 과정을 보여준다.

그림 10-9. 실례 10-13의 조작

실례 10-13은 지적자의 능력을 시험한다. 지적자는 특정한 함수가 이름이 알려지지 않은 변수들과 배렬원소들에 대하여 조작하는 일관성있고 효과적인 방법을 준다.

제 4 절. 지적자와 C형문자렬

7장에서 언급하였지만 C형문자렬은 단순히 char형의 배렬이다. 지적자표기를 배렬원소들에 적용할수 있는것처럼 문자렬안의 문자들에도 적용할수 있다.

1. 문자렬상수에로의 지적자

실례 10-14는 두개의 문자렬을 정의하는데 하나는 배렬표기, 다른 하나는 지적자표기를 사용한다.

(실례 10-14) 배렬과 지적자표기를 사용하여 문자렬 정의

```
#include <iostream>
using namespace std;
int main()
{
   char str1[] = "Defined as an array";
   char* str2 = "Defined as a pointer";
   cout << str1 << endl;
   cout << str2 << endl;
// str1++;      // 오유: str1은 상수이다.
   str2++;      // 옳다: str2은 지적자이다.
   cout << str2 << endl;
   return 0;
}
```

두가지 형식의 정의는 여러 측면에서 비슷하다. 실례에서 보여주는것처럼 두개 문자렬을 출력할수 있고 그것들을 함수인수로 사용할수 있다. 그러나 약간 차이가 있다. str1은 주소 즉 지적자상수이다. 그러나 str2은 지적자변수이다. 따라서 str2은 변경할

수 있으나 str1은 변경할수 없다. 그림 10-10은 기억기에서 두 종류의 문자렬을 고찰하고있다.

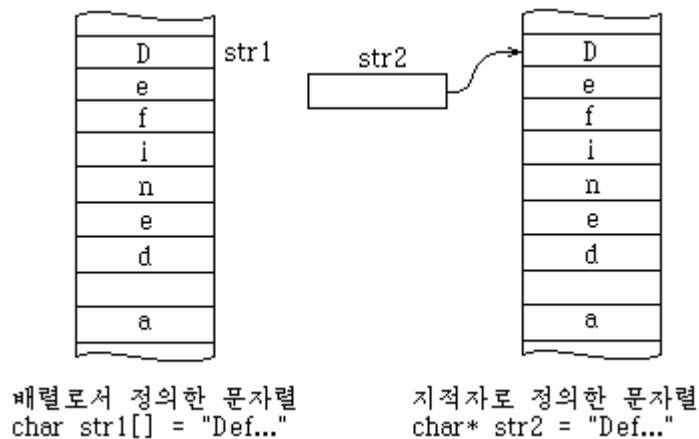



그림 10-10. 배렬과 지적자로서의 문자렬

str2은 지적자이므로 증가시킬수 있으나 일단 증가되면 문자렬의 첫 원소를 더는 가리키지 않는다. 여기에 실례 10-14의 출력이 있다.

```
Defined as an array
Defined as a pointer
fined as a pointer ← str2++후의 명령문
```

지적자로 정의한 문자렬은 배렬로 정의한것보다 훨씬 더 융통성이 있다. 다음의 실례들은 그 융통성을 사용한다.

2. 함수인수로서의 문자렬

여기에 문자렬을 함수인수로 사용하는 실례가 있다. 함수는 매개 문자를 하나씩 차례로 호출하는 방법으로 단순히 문자렬을 출력한다.

(실례 10-15) 지적자표기를 사용하여 문자렬 표시

```
#include <iostream>
using namespace std;
void DispStr(char*);
int main()
{
   char str[] = "이것은 문자렬이다.";
   DispStr(str);
   return 0;
}
void DispStr(char* ps)
{
   while(*ps)
      cout << *ps++;
   cout << endl;
}
```

배렬주소 str는 함수 DispStr()의 호출에서 인수로 쓰인다. 배렬주소는 상수이지만 값에 의해 넘겨지므로 그 사본이 DispStr()에 창조된다. 이 사본은 지적자 ps이다. 지적자를 변경할수 있으므로 함수는 ps를 증가시키면서 문자렬을 표시한다. 식 *ps++는 문자렬의 문자들을 차례로 돌려준다. 순환은 문자렬끝에 있는 null문자('\0')를 검출할 때까지 반복된다. null이 false를 표시하는 값 0을 가지므로 while순환은 여기서 끝난다.

3. 지적자를 사용한 문자렬 복사

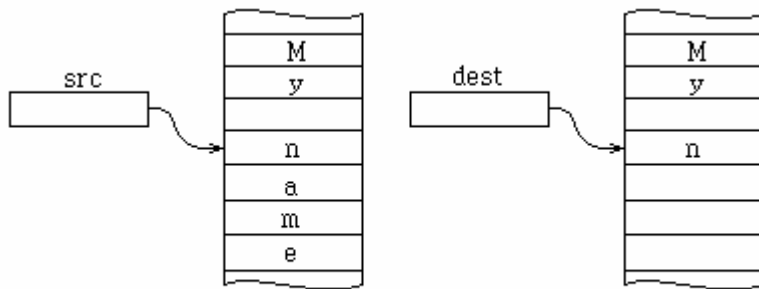
배렬로부터 값을 얻는데 지적자를 사용하는 실례를 보았다. 또한 지적자는 배렬에 값을 삽입하는데 사용할수 있다. 실례 10-16은 한 문자렬을 다른 문자렬에 복사하는 함수를 보여준다.

(실례 10-16). 지적자를 사용한 문자렬 복사

```
#include <iostream>
using namespace std;
void CopyStr(char*, const char*);
int main()
{
  char* str1 = "이것은 문자렬이다.";
  char str2[80];
  CopyStr(str2, str1);
  cout << str2 << endl;
  return 0;
}
void CopyStr(char* dest, const char* src)
{
  while(*src)
    *dest++ = *src++;
  *dest = '\0';
}
```

여기서 main()에서는 함수 CopyStr()를 호출하여 str1을 str2에 복사한다. 이 함수에서 식 *dest++=*src++는 src가 가리키는 주소의 값을 가지며 dest가 가리키는 주소에 값을 보관한다. 그다음 두 지적자는 증가되고 다음 문자가 전송된다. 순환은 src에서 null문자가 검출될 때 끝나고 여기서 null을 dest에 삽입하고 함수는 귀환한다. 그림 10-11은 문자렬을 처음부터 끝까지 순환하는 방법을 보여준다.

그림 10-11. 실례 10-16의 조작

4. 서고문자렬함수

이미 문자렬에 사용한 많은 서고함수들은 지적자표기에 의하여 지적되는 문자렬인수를 가진다. 번역프로그람의 문서(혹은 string.h머리부파일)에 strcpy()의 선언을 보여주는 실례가 있다. strcpy()함수는 한 문자렬을 다른 문자렬에 복사하는데 이것을 자체로 만든 CopyStr()함수와 비교할수 있다. strcpy()서고함수의 문법은

```
char* strcpy(char* dest, const char*src);
```

strcpy()함수는 char*형의 인수를 두개 가진다. strcpy()함수도 역시 char에로의 지적자를 돌려주며 이것은 dest문자렬의 주소이다. 이 함수는 자체로 만든 CopyStr()함수와 비슷하게 동작한다.

5. const변경자와 지적자

const변경자와 지적자선언자의 사용을 혼돈하기 쉽다. 다음의 명령문들은 두가지 가능성을 보여준다.

```
const int* cptrInt; // cptrInt는 상수int에로의 지적자
int* const ptrcInt; // ptrcInt는 int에로의 상수지적자
```

첫째 행의 선언후에 cptrInt자체는 변경할수 있으나 cptrInt가 가리키는 값은 변경할수 없다. 둘째 행의 선언후에 ptrcInt가 가리키는 값은 변경할수 있으나 ptrcInt자체의 값은 변경할수 없다. 설명문에 보여준것처럼 오른쪽으로부터 왼쪽으로 읽어서 그 차이를 기억할수 있다. 지적자를 만들 때와 그것이 상수를 가리킬 때 두 위치에서 const를 사용할수 있다.

strcpy()의 선언에서 인수 const char* src는 src가 가리키는 문자렬을 strcpy()에서 변경할수 없다는것을 지적한다. 이것은 src지적자자체를 변경할수 없다는것을 암시하지 않는다. 지적자자체를 변경하려면 인수선언을 char* const src로 하여야 한다.

6. 문자렬에로의 지적자들의 배렬

int형이나 float형변수들의 배렬이 있듯이 지적자배렬도 있다. 지적자배렬의 일반적

인 사용으로서 문자렬에로의 지적자들의 배렬이 있다. 실례 7-18에서 char*문자렬의 배렬을 보여주었다. 문자렬의 배렬을 사용하면 결함이 있다. 즉 문자렬들을 보관하는 보조배렬들이 모두 같은 길이를 가져야 하므로 문자렬이 보조배렬의 길이보다 짧으면 기억공간이 랑비된다. 지적자를 사용하여 이 문제를 해결하는 방법을 고찰하자. 실례 7-18을 변경하여 문자렬의 배렬이 아니라 문자렬에로의 지적자들의 배렬을 창조하자. 여기에 실례 10-17이 있다.

(실례 10-17) 문자렬에로의 지적자배렬

```
#include <iostream>
using namespace std;
const int DAYS = 7;
int main()
{
  char* arrPtrs[DAYS] = { "Sunday", "Monday", "Tuesday", "Wednesday",
            "Thersday", "Friday", "Saterday" };

  for(int j=0; j<DAYS; j++)
    cout << arrPtrs[j] << endl;
  return 0;
}
```

이 프로그람의 출력은 실례 7-18과 같다.

```
Sunday
Monday
Tuesday
Wednesday
Thersday
Friday
Saterday
```

문자렬들이 배렬의 부분이 아닐 때 C++는 그것들을 기억기에 련이어 배치하므로 기억공간을 랑비하지 않는다. 그러나 문자렬을 찾으려면 문자렬에로의 지적자들을 보관하는 배렬이 있어야 한다. 문자렬자체는 char형의 배렬이므로 문자렬에로의 지적자배렬은 char에로의 지적자들의 배렬이다. 즉 실례 10-17에서 arrPtrs정의의 의미와 같다.

문자렬은 항상 유일한 주소로 표시되는데 그 주소는 문자렬의 첫 문자의 주소이고 배렬안에 보관되는 주소이다. 그림 10-12는 이것을 보여준다.

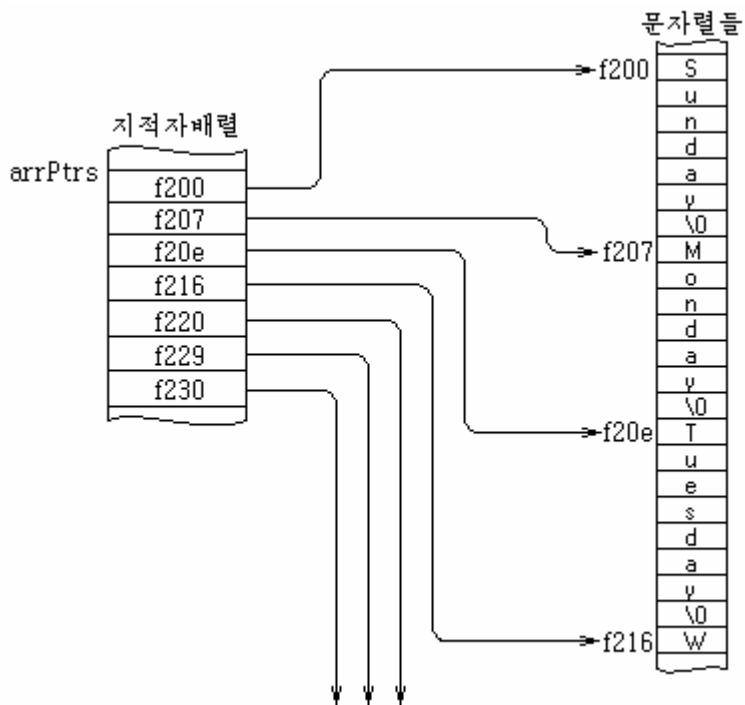



그림 10-12. 지적자와 문자렬들의 배렬

제 5 절. new와 delete에 의한 기억기관리

기억기를 확보하는데 배렬을 사용하는 실례를 많이 보았다. 다음 명령문은 기억기에 100개의 옹근수를 확보한다.

```
int arr[100];
```

배렬은 자료를 보관하는 쓸모있는 수법이다. 그러나 배렬에는 일련의 제한이 있다. 그것은 프로그람을 작성할 때 배렬의 크기를 알아야 하는것이다. 배렬의 크기를 지적하기 위해 프로그람을 실행할 때까지 기다릴수 없다. 즉 다음의 수법은 동작하지 않는다.

```
cin >> size;
int arr[size]; // 오유: 배렬의 크기는 상수이여야 한다.
```

번역프로그람은 배렬크기가 상수이여야 한다는 오유를 출력한다.

그러나 많은 경우에 실행시에 기억기가 얼마나 요구되는지 모른다. 실례로 사용자가 입력한 문자렬을 보관하려고 할 때 기대하는 최대문자렬을 보관하는데 알맞는 크기를 가지는 배렬을 정의할수 있지만 이것은 기억기를 랑비한다.

1. new연산자

C++는 기억블로크를 얻는 다른 방법으로서 new연산자를 제공한다. new연산자는 조작체계로부터 기억기를 얻어서 그 선두에로의 지적자를 돌려준다. 실례 10-18은

new의 사용법을 보여준다.

(실례 10-18) new연산자

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
   char* str = "이것은 문자렬이다.";
   int len = strlen(str);
   char* ptr;
   ptr = new char[len + 1];
   strcpy(ptr, str);
   cout << "ptr=" << ptr << endl;
   delete[] ptr;
   return 0;
}
```

식

```
ptr = new char[len+1];
```

은 문자렬 str를 보관하는데 충분한 기억공간에로의 지적자를 돌려주며 그 길이 len은 strlen()서고함수의 값에다가 문자렬끝 '\0'문자를 위한 여유바이트를 더한 값이다. 그림 10-13은 new연산자를 사용하는 명령문의 문법을 보여준다. 크기의 량끝에 괄호를 쓴다.

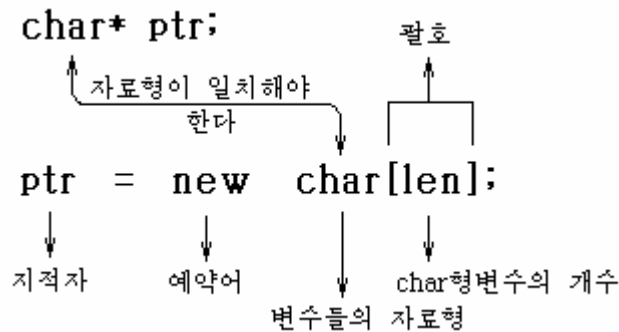



그림 10-13. new연산자의 문법

그림 10-14는 new에 의해 얻어진 기억기와 그것을 가리키는 지적자를 보여준다.

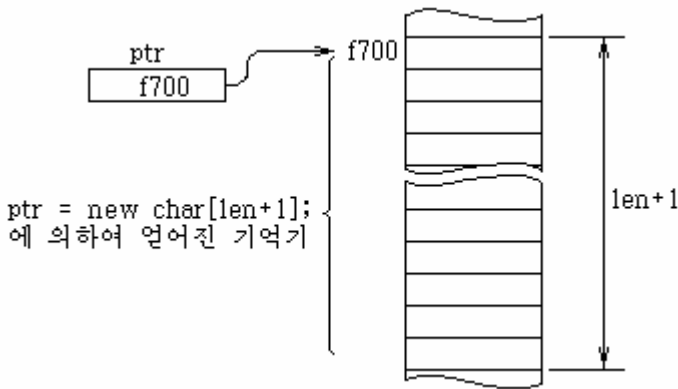



그림 10-14. new연산자에 의해 얻은 기억기

실례 10-18에서는 ptr가 가리키는 새로 창조된 기억령역에 문자렬 str를 복사할 때 strcpy()를 사용한다. 이 령역은 str의 길이와 같으므로 문자렬을 정확히 복사한다.

실례 10-18의 출력은 다음과 같다.

이것은 문자렬이다.

C프로그람작성자들은 new가 서고함수 malloc()계렬과 비슷한 역할을 한다는것을 알고있다. new는 적당한 자료형에로의 지적자를 돌려주지만 malloc()의 지적자는 적당한 형으로 강제형변환해야 한다. 물론 다른 우점도 있다.

C프로그람작성자들은 이미 할당한 기억기의 크기를 변경하는 realloc()와 등가한 기능이 C++에 있었으면 한다. 그러나 C++에는 renew가 없다. 따라서 new를 사용하여 더 큰(작은) 공간을 창조하고 이전 령역에서 새 령역에로 자료를 복사해야 한다.

2. delete연산자

new를 사용하여 기억기의 많은 블로크를 예약하면 우연히 모든 유효기억기를 예약하게 되여 체계가 파괴된다. 기억기의 안전하고 효과적인 사용을 담보하기 위하여 new연산자와 조작체계에 기억기를 돌려주는 delete연산자를 함께 사용한다.

실례 10-18에서 명령문

```
delete[] ptr;
```

은 체계에 ptr가 가리키는 기억기를 돌려준다.

실제로 실례 10-18에는 이 연산자가 필요없다. 그것은 프로그람이 완료될 때 기억기를 자동적으로 돌려주기때문이다. 그러나 함수에서 new를 사용하는 경우를 가정하자. 함수가 그 기억기에로의 지적자로서 국부변수를 사용한다면 함수가 완료할 때 지적자는 해체되지만 기억기는 남아있고 프로그람에서 호출할수 없는 공간이 만들어진다. 그러므로 delete를 사용하여 기억기를 해방하는것이 중요하다.

기억기의 해방은 그것을 가리키는 지적자를 삭제하지 않으며 지적자의 변수값을 변경하지 않는다. 그러나 그 주소는 더는 유효한 주소가 아니고 그것이 가리키는 기억

기는 완전히 다른것으로 변경될수 있다. 따라서 해방된 기억기에로의 지적자를 사용하지 않도록 주의해야 한다.

delete뒤의 중괄호는 그것이 배렬을 해방한다는것을 가리킨다. new를 사용하여 하나의 객체를 창조한다면 그것을 해방할 때 괄호는 필요없다.

```
ptr = new someclass;      // 하나의 객체를 할당한다.
…
delete ptr;               // delete뒤에 괄호가 없다.
```

3. new를 사용하는 문자렬클라스

new연산자는 때때로 구성자에 나타난다. 실례 8-6에서 마지막으로 본 String클라스를 변경하자. 모든 String객체는 꼭같은 크기의 고정기억기를 차지한다.

이 고정길이보다 짧은 문자렬은 기억기를 랑비하고 긴 문자렬은 배렬끝을 벗어나게 함으로써 체계를 파괴할수 있다. 다음 실례는 new에 의하여 정확한 기억량을 얻는다.

(실례 10-19) new를 사용하여 문자렬용 기억기얻기

```
#include <iostream>
#include <cstring>
using namespace std;
class String
{
private:
   char* str;
public:
   String(char* s)
   {
      int length = strlen(s);
      str = new char[length + 1];
      strcpy(str, s);
   }
   ~String()
   {
      delete[] str;
   }
   void Display()
   {
      cout << str << endl;
   }
};
int main()
{
   String s1 = "이것은 문자렬이다.";
   cout << "s1=";
   s1.Display();
   return 0;
```

```
}
```

String클라스는 한개 자료항목 즉 str라고 부르는 char에로의 지적자를 가진다. 이 지적자는 String객체에 의해 유지되는 문자렬을 가리킨다. 객체안에는 문자렬을 유지하기 위한 어떤 배렬도 없으며 그에 대한 지적자가 String의 유일한 성원이다.

- 실례 10-19의 구성자

이 실례의 구성자는 표준 char*문자렬을 인수로 가진다. 구성자는 new를 사용하여 문자렬용의 기억공간을 얻고 str는 새로 얻은 기억기를 가리킨다. 그다음 구성자는 strcpy()에 의하여 새 공간에 문자렬을 복사한다.

- 실례 10-19의 해체자

지금까지의 실례들에서 구성자를 많이 보았는데 new에 의하여 기억기를 할당하는 경우에는 해체자가 매우 중요하다. 객체를 창조할 때 기억기를 할당하면 그 객체가 더는 필요하지 않을 때 기억기를 해방하는것이 합리적이다. 해체자는 객체가 해체될 때 자동적으로 호출되는 함수이다. 실례 10-19의 해체자는 다음과 같다.

```
~String()
{
   delete[] str;
}
```

이 해체자는 객체가 창조될 때 얻어진 기억기를 체계에 돌려준다. 객체는 그것이 정의된 함수가 완료될 때에 대체로 해체된다. 이처럼 해체자는 String객체에 의해 얻어진 기억기를 체계에 돌려준다는것을 담보한다.

실례 10-19에 보여준것처럼 해체자를 사용하는데서 그 어떤 사소한 결함도 있어서는 안된다. 만일 어떤 String객체를 다른 객체에 복사한다면, 즉 s3=s1과 같은 명령문을 사용한다면 실제로는 문자렬에로의 지적자(char*)만 복사된다. 그러면 두 객체는 기억기의 같은 문자렬을 가리키게 된다. 이때 한개 문자렬을 해방하면 해체자는 char 문자렬을 해방하므로 다른 객체에는 무효한 지적자만 남는다. 이러한 오유는 국부객체가 함수에서 귀환할 때처럼 객체들이 명백하지 않은 방법으로 해체되므로 포착하기 힘들다.(2장 참고)

제 6 절. 객체에로의 지적자

지적자는 단순자료형과 배렬은 물론 객체를 지적할수 있다.

실례들에서 이름을 주어서 정의한 객체를 많이 보았다. 례를 들면

```
Distance dist;
```

여기서 dist라는 객체는 Distance클라스형으로 정의된다.

그러나 때때로 프로그람을 쓸 때 몇개의 객체를 창조할지 모르는 경우가 있다. 이것은 프로그람을 실행할 때 new를 사용하여 객체를 창조하는 경우이다. 이미 알고있

는것처럼 new는 이름없는 객체에로의 지적자를 돌려준다.

그러면 객체를 창조하는 두가지 방법을 대비하는 간단한 실례 10-20을 고찰하자.

(실례 10-20). 지적자에 의한 성원함수호출

```
#include <iostream>
using namespace std;
class Distance
{
protected:
   int meters;
   float centies;
public:
   void GetDist()
   {
      cout << "\n메터를 입력하십시오:";
      cin >> meters;
      cout << "센치메터를 입력하십시오:";
      cin >> centies;
   }
   void ShowDist()
      { cout << meters << "m " << centies << "cm"; }
};
int main()
{
   Distance dist;
   dist.GetDist();
   dist.ShowDist();
   Distance* distPtr;
   distPtr = new Distance;
   distPtr->GetDist();
   distPtr->ShowDist();
   cout << endl;
   return 0;
}
```

이 프로그람에서는 9장에서 본 Distance를 사용한다.

main()함수에서 dist를 정의하고 Distance성원함수 GetDist()에 의하여 사용자로부터 거리를 얻고 ShowDist()에 의하여 표시한다.

1. 성원의 참고

실례 10-20은 new연산자를 사용하여 Distance형의 객체를 창조하고 그 주소를 distPtr라는 지적자에 돌려준다.

여기서 distPtr가 지적하는 객체에서 성원함수를 어떻게 호출하겠는가?

점성원호출연산자를 사용할수 있으나 그것은 동작하지 않는다. 즉

```
distPtr.GetDist(); // 오유: distPtr는 변수가 아니다.
```

점연산자는 그 왼변에 변수를 요구한다. distPtr는 변수에로의 지적자이므로 다른 문법을 사용할수 있다. 한가지 수법은 그 지적자를 참고하는것이다. 즉 지적자가 가리키는 변수의 내용을 얻는다.

```
(*distPtr).GetDist();
```

그러나 괄호가 있으므로 쓰기 불편하다. 괄호는 점연산자가 간접연산자(*)보다 높은 우선순위를 가지므로 필요하다.

더 좋은 방법으로서 성원호출연산자를 사용하는 방법이 있다.

```
distPtr->GetDist();
```

실례 10-20에서 알수 있는것처럼 ->연산자는 점연산자가 객체들과 작업하는것과 같은 방법으로 객체에로의 지적자들과 작업한다. 여기에 프로그람의 출력이 있다.

```
메터를 입력하십시오: 10
센치메터를 입력하십시오: 6.25
10m 6.25cm
메터를 입력하십시오:6
센치메터를 입력하십시오:4.25
6m 4.25cm
```

2. new의 다른 사용방법

객체의 기억기를 얻는데 new를 사용하는 수법이 있다.

new는 객체를 보관하는 기억령역에로의 지적자를 돌려주므로 지적자를 간접참고하여 원시객체를 참고할수 있다. 실례 10-21은 그 방법을 보여준다.

(실례 10-21) new에 의해 돌아온 지적자의 비참고

```
#include <iostream>
using namespace std;
class Distance
{
private:
   int meters;
   float centies;
public:
   void GetDist()
   {
      cout << "\n메터를 입력하십시오:";
      cin >> meters;
      cout << "센치메터를 입력하십시오:";
      cin >> centies;
   }
   void ShowDist()
      { cout << meters << "m " << centies << "cm"; }
};
int main()
{
   Distance& dist = *(new Distance);
```

```
    dist.GetDist();
    dist.ShowDist();
    cout << endl;
    return 0;
}
```

식 new Distance는 Distance객체에 충분한 기억기에로의 지적자를 돌려주므로 원시객체를 다음과 같은 방법으로 호출할수 있다.

```
*(new Distance)
```

이것은 지적자가 가리키는 객체이다. 참고를 사용하여 dist를 Distance형객체로 정의하고 그것을 *(new Distance)로서 설정한다. 그러면 ->가 아니라 점성원호출연산자에 의하여 dist의 성원을 참고할수 있다.

이 방법은 new에 의해 얻어진 객체를 가리키는 지적자에 의하여 객체를 선언하므로 일반성이 적지만 객체를 사용할 때와 같은 방법으로 성원함수들을 호출한다.

3. 객체에로의 지적자배렬

객체에로의 지적자배렬은 프로그람작성의 일반적인 구성요소이다.

이러한 배렬은 객체들의 묶음을 호출할수 있게 하므로 배렬에 객체자체를 배치하는것보다 더 융통성이 있다.(실례 10-13에서 객체들의 지적자배렬을 정의함으로써 객체들을 정렬하지 않고도 지적자를 리용하여 정렬할수 있다는것을 보았다.)

(실례 10-22) 객체에로의 지적자배렬

```
#include <iostream>
using namespace std;
class Person
{
protected:
    char name[40];
public:
    void SetName()
    {
        cout << "\n이름을 입력하십시오:"; cin >> name;
    }
    void PrintName() { cout << "\n 이름=" << name; }
};
int main()
{
    Person* persPtr[100];
    int n = 0;
    char choice;
    do
    {
        persPtr[n] = new Person;
        persPtr[n]->SetName();
        n++;
```

```
        cout << "계속하겠습니까(y/n)? "; cin >>  choice;
    } while(choice == 'y');
    for(int j=0; j<n; j++) {
        cout << "\n개인자료번호 " << j + 1;
        persPtr[j]->PrintName();
    }
    cout << endl;
    return 0;
}
```

클라스 Person은 하나의 자료항목으로서 사람의 이름을 포함하는 문자렬 name을 가진다. 또한 두개의 성원함수 SetName()과 PrintName()은 이름을 설정하고 표시한다.

1) 프로그람의 조작

main()함수는 Person형의 지적자를 100개 가지는 배렬 persPtr를 정의한다. do순환에서 사용자가 이름을 입력하면 이름에 기초하여 new에 의해 Person객체를 창조하고 그 객체에로의 지적자를 배렬 persPtr에 보관한다. 지적자에 의하여 객체를 쉽게 호출할수 있다는것을 보여주기 위하여 여러개의 Person객체의 name자료를 출력한다. 프로그람과의 대화는 다음과 같다.

```
이름을 입력하십시오:  김철호
계속하겠습니까(y/n)? y
이름을 입력하십시오:  강철
계속하겠습니까(y/n)? y
이름을 입력하십시오:  조순애
계속하겠습니까(y/n)? n
개인자료번호 1
이름=김철호
개인자료번호 2
이름=강철
개인자료번호 3
이름=조순애
```

2) 성원함수호출

배렬 perPtr에서 지적자들이 가리키는 Person객체들의 성원함수 SetName()과 PrintName()을 호출해야 한다. 배렬 perPtr의 매개 원소는 배렬표기 persPtr[j]에 의해 지적한다. 이것은 *(persPtr+j)라는 지적자표기와 등가하다.

배렬원소들은 Persone형의 객체에로의 지적자이다. 지적자에 의하여 객체의 성원을 호출할 때 ->연산자를 사용한다. 즉 GetName()에 대하여

```
persPtr[j]->GetName();
```

이것은 persPtr배렬의 원소 j에 의해 지적된 Person객체의 성원함수 GetName()을 실행한다.

제 7 절. 련결목록실례

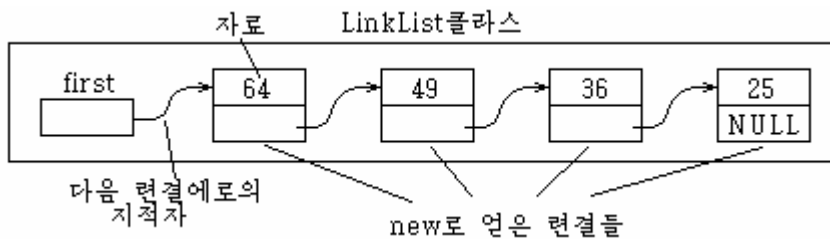
다음 실례는 간단한 련결목록을 보여준다. 련결목록은 자료를 보관하는 다른 한가지 방법이다. 이미 배렬에 자료를 보관하는 실례를 많이 보았다. 또한 실례 10-17과 실례 7-22에서와 같이 자료성원에로의 지적자배렬도 사용하였다. 배렬과 지적자배렬은 모두 프로그람을 실행하기 전에 고정크기배렬을 선언해야 한다.

1. 지적자들의 사슬

련결목록은 배렬을 전혀 사용하지 않는 융통성있는 보관체계를 제공한다. 그대신 배렬의 자료항목용 공간을 필요할 때마다 new에 의해 얻으며 매개 항목은 지적자를 통하여 다음 자료항목과 련결된다. 개별적인 항목들은 배렬원소들처럼 기억기의 련속위치에 놓을 필요가 없으며 그것들은 도처에 분산된다.

실례에서 련결목록은 클라스 LinkList형의 객체이다. 개별적인 자료항목이나 련결은 Link형의 구조체에 의해 표시된다. 매개 구조체는 유일한 자료항목을 포함하는 옹근수와 다음 련결에로의 지적자를 포함한다. 목록자체는 목록의 머리부에 있는 련결에로의 지적자를 보관한다. 련결목록의 구조를 그림 10-15에 주었다.

그림 10-15. 련결목록

(실례 10-23) 련결목록

```
#include <iostream>
using namespace std;
struct Link
{
   int data;
   Link* next;
};
class LinkList
{
private:
   Link* first;
public:
   LinkList() { first = NULL; }
   void AddItem(int);
   void Display();
```

```
};
void LinkList::AddItem(int d)
{
  Link* newLink = new Link;
  newLink->data = d; newLink->next = first;
  first = newLink;
}
void LinkList::Display()
{
  Link* current = first;
  while(current != NULL)
  {
     cout << current->data << endl;
     current = current->next;
  }
}
int main()
{
   LinkList li;
   li.AddItem(25);
   li.AddItem(36);
   li.AddItem(49);
   li.AddItem(64);
   li.Display();
   return 0;
}
```

LinkList클라스는 유일한 성원자료항목 즉 목록선두에로의 지적자를 가진다. 목록이 창조될 때 구성자는 이 지적자 first를 NULL로 초기화한다. 상수 NULL는 0으로 정의되여있다. 이 값은 지적자가 유효주소를 보관하지 않는다는것을 경고한다. 프로그람에서 다음 성원이 NULL을 가지는 련결을 목록의 끝으로 가정한다.

2. 목록에 항목의 추가

AddItem()성원함수는 련결목록에 항목을 추가한다. 이때 새로운 련결은 목록의 선두에 삽입된다. AddItem()함수를 목록의 끝에 항목들을 삽입하는것으로 만들수도 있으나 그러면 프로그람은 더 복잡해진다.

그러면 새로운 련결을 삽입하는 단계들을 고찰하자.

우선 Link형의 새로운 구조체를 창조한다.

```
Link* newLink = new Link;
```

이것은 new로 새로운 Link용 기억기를 창조하고 newLink변수에 기억기에로의 지적자를 보관한다.

다음에 새로 창조한 구조체의 성원들에 적당한 값을 설정한다. 구조체는 클라스와 비슷하므로 이름이 아니라 지적자에 의해 참고할 때 그 성원은 ->성원호출연산자에

의하여 호출할수 있다.

다음의 두 행은 data변수를 AddItem()에 넘어온 인수값으로 설정하고 next지적자를 목록의 선두에로의 지적자를 보관하는 first로 설정한다.

```
newLink->data = d;
newLink->next = first;
```

끝으로 새로운 련결을 지적하도록 first변수를 설정한다.

```
first = newLink;
```

그리하여 first와 낡은 first련결사이의 련계를 끊고 새로운 련결을 삽입하고 낡은 first련결을 두번째 위치로 옮긴다. 그림 10-16에서 그 과정을 보여준다.

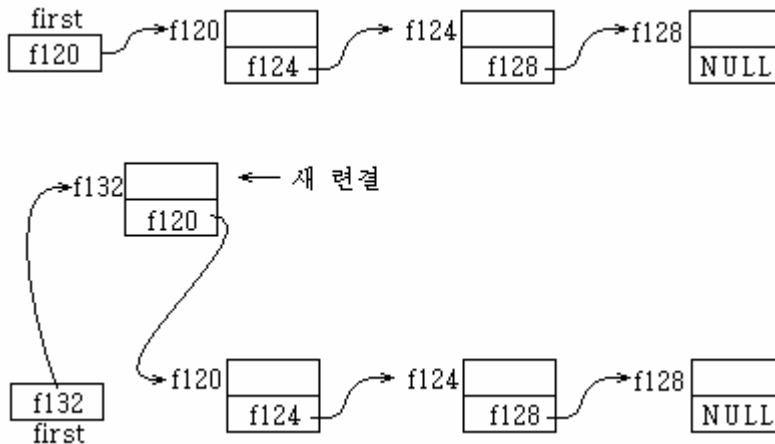



그림 10-16. 련결목록에 추가

3. 목록내용의 표시

일단 목록을 창조하면 모든 성원들을 하나씩 이동하면서 그것들을 표시하기 쉽다. 그러자면 next지적자가 목록의 끝을 알려주는 NULL일 때까지 지적자를 한 련결에서 다른 련결에로 이동하여야 한다. 함수 Display()에서 다음의 행

```
cout << endl << current->data;
```

는 자료의 값을 출력하며

```
current = current->next;
```

는 while식의 current != NULL이 거짓으로 될 때까지 한 련결로부터 다른 련결에로 이동한다. 여기에 실례 10-23의 출력이 있다.

```
64
47
36
29
```

련결목록은 배렬다음으로 가장 널리 쓰이는 자료보관수법이다. 이미 언급한것처럼 련결목록은 배렬에 의해 발생하는 기억공간의 랑비를 피한다. 결함은 련결목록우에 있는 어떤 원소를 검색하려면 목록의 선두로부터 필요한 련결에 이를 때까지 련결들의

사슬을 항행해야 하는것이다. 이것은 시간을 소비한다. 그러나 배렬원소는 첨수에 의해 고속으로 호출할수 있다.

4. 자기를 포함하는 클라스

자기를 포함하는 클라스와 구조체를 사용할수 있다. 실례 10-23의 Link구조체는 같은 종류의 구조체에로의 지적자를 포함한다.

```
class SampleClass
{
   SampleClass* ptr;
};
```

클라스는 자기형의 객체에로의 지적자를 포함할수 있지만 자기 형의 객체를 포함할수 없다.

```
class SampleClass
{
   SampleClass obj;    // 오유
};
```

이것은 클라스는 물론 구조체인 경우도 같다.

5. LinkList의 확장

련결목록의 구성은 실례 10-23에서 보여준것보다 더 복잡할수 있다. 매개 련결에는 더 많은 자료가 있을수 있다. 련결에는 옹근수대신 많은 자료항목이나 구조체 또는 객체에로의 지적자를 보관할수 있다.

성원함수들을 추가하여 사슬의 임의의 부분으로부터 련결을 추가, 삭제하는 기능도 실현할수 있다. 다른 중요한 성원함수는 해체자이다. 이미 언급한것처럼 더는 필요없는 기억블로크를 삭제하는것이 중요하다. 바로 해체자가 이 작업을 수행하므로 LinkList클라스에 대단히 필요하다. 매개 련결이 차지한 기억기를 해방하기 위하여 delete를 사용하면서 목록을 횡단할수 있다.

제 8 절. 지적자에로의 지적자

다음 실례는 객체에로의 지적자배렬을 보여주며 객체의 자료에 기초하여 지적자들을 정렬하는 방법을 보여준다. 실례는 지적자에로의 지적자를 포함한다.

다음 프로그람에서 기본은 Person클라스의 객체에로의 지적자배렬을 창조하는것이다. 이것은 실례 10-22와 비슷하지만 실례 10-24의 Order()와 BSort()들을 변경하고 이름의 자모순에 기초하여 Person객체들의 묶음을 분류한다. 여기에 실례 10-24의 프로그람이 있다.

(실례 10-24) 지적자배렬을 사용한 개인자료의 정렬

```
#include <iostream>
#include <string>
using namespace std;
class Person
{
protected:
  string name;
public:
  void SetName()
  {
     cout << "\n이름을 입력하십시오:";
     cin >> name;
  }
  void PrintName() { cout << endl << name; }
  string GetName() { return name; }
};
void BSort(Person**, int);
void Order(Person**, Person**);
int main()
{
  Person* persPtr[100];
  int n = 0;
  char choice;
  do
  {
     persPtr[n] = new Person;
     persPtr[n]->SetName();
     n++;
     cout << "계속하겠습니까(y/n)? ";
     cin >> choice;
  } while(choice == 'y');
  cout << "\n정렬하지 않은 목록; ";
  for(int j=0; j<n; j++)
     persPtr[j]->PrintName();
  cout << endl;
  BSort(persPtr, n);
  cout << "\n정렬된 목록; ";
  for(int j=0; j<n; j++)
     persPtr[j]->PrintName();
  cout << endl;
  return 0;
}
void BSort(Person** pp, int n)
{
  int j, k;
  for(j=0; j<n-1; j++)
     for(k=j+1; k<n; k++)
         Order(pp + j, pp + k);
```

```
}
void Order(Person** pp1, Person** pp2)
{
  if((*pp1)->GetName() > (*pp2)->GetName())
  {
     Person* tempPtr = *pp1;
     *pp1  = *pp2;
     *pp2 = tempPtr;
  }
}
```

프로그람은 실행할 때 우선 이름을 요구한다. 사용자가 이름을 하나 입력하면 Person형의 객체를 하나 창조하고 그 객체의 name자료를 사용자가 입력한 이름으로 설정한다. 또한 프로그람은 persPtr배렬에 객체에로의 지적자를 보관한다.

사용자가 이름을 더는 입력하지 않으려고 n을 입력하면 프로그람은 BSort()함수를 호출하여 Person객체의 name성원변수에 기초하여 객체들을 정렬한다. 여기에 대화가 있다.

```
이름을 입력하십시오:  김철호
계속하겠습니까(y/n)? y
이름을 입력하십시오:  강철
계속하겠습니까(y/n)? y
이름을 입력하십시오:  차혁철
계속하겠습니까(y/n)? y
이름을 입력하십시오:  조순애
계속하겠습니까(y/n)? n
개인자료번호 1
이름=김철호
개인자료번호 2
이름=강철
개인자료번호 3
이름=조순애
정렬하지 않은 목록;
김철호
강철
차혁철
조순애
정렬된 목록;
강철
김철호
조순애
차혁철
```

1. 지적자정렬

실제로 Person객체들을 정렬할 때 객체자체를 옮기지 않고 객체에로의 지적자를 옮긴다. 이것은 객체가 큰 경우 시간을 많이 소비하게 하는 기억기안에서의 객체의 이

동을 방지하게 한다. 그 처리를 그림 10-17에서 보여준다.

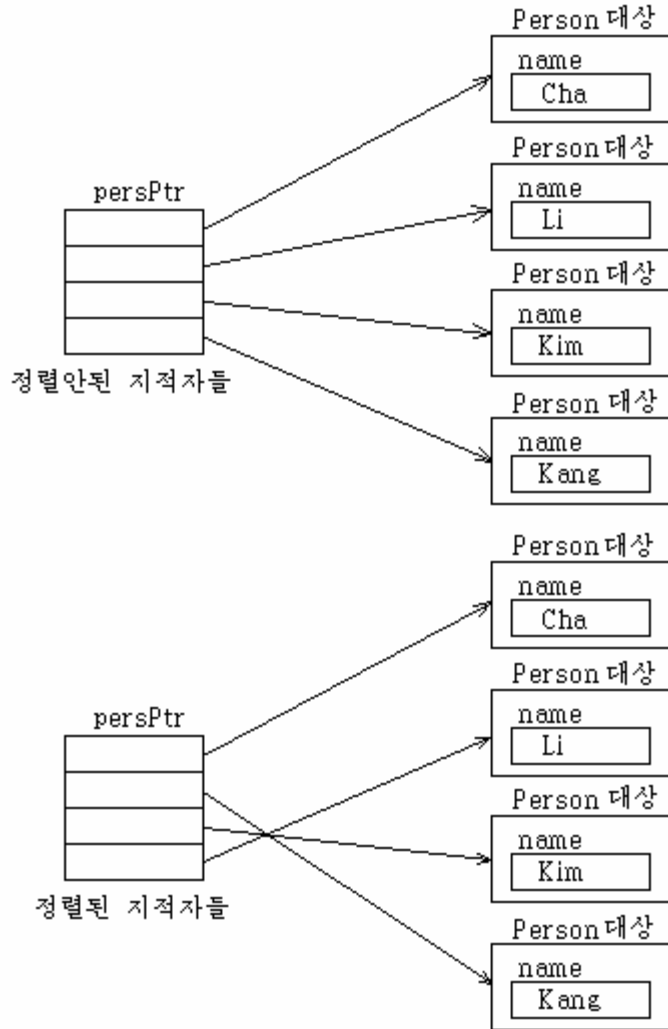



그림 10-17. 지적자배렬의 정렬

정렬작업이 편리하도록 Person클라스안에 GetName()성원함수를 추가하고 지적자들을 교체해야 할 경우를 찾기 위하여 Order()로부터 이름을 호출하게 하였다.

2. Person**자료형

BSort()함수의 첫 인수와 Order()의 두 인수는 Person**형을 가진다.

그러면 여기서 두개의 별표는 무엇을 의미하는가?

인수들은 배렬 persPtr의 주소를 넘기는데 사용되며 Order()의 경우에는 배렬원소들의 주소를 넘기는데 사용된다. 이것이 Person형의 배렬이라면 배렬의 주소는 Person*형이다. 그러나 배렬은 Person에로의 지적자형 또는 Person*형이므로 그 주소는 Person**형이다. 지적자의 주소는 지적자에로의 지적자이다. 그림 10-18에서 이것을 보여준다.

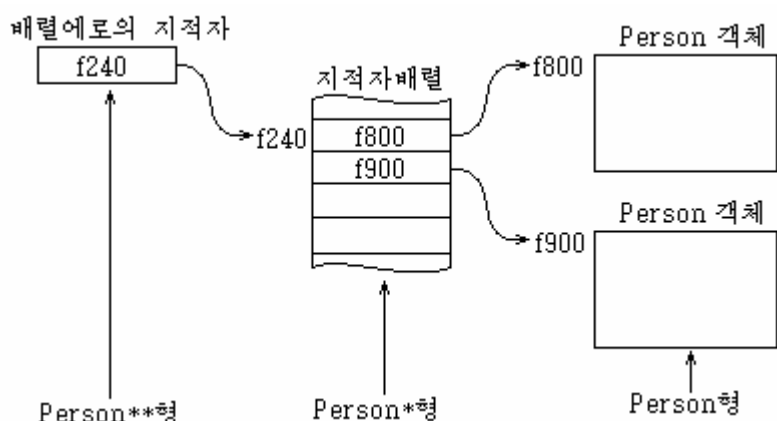



그림 10-18. 지적자배렬에로의 지적자

이 프로그람을 int형배렬을 정렬하는 실례 10-13과 비교하자. 실례 10-24에서는 함수에 넘긴 자료형들이 모두 실례 10-13에서보다 하나 더 많은 별표를 가진다. 이것은 그 배렬이 지적자배렬이기때문이다.

Persptr배렬이 지적자를 포함하므로

```
persPtr[j]->PrintName();
```

은 persPtr의 원소 j에 의해 지적된 객체의 PrintName()함수를 실행한다.

3. 문자렬비교

실례 10-24의 Order()함수를 두개의 문자렬을 자모순으로 비교할수 있게 수정한다. 여기서는 C++서고함수 strcmp()에 의하여 문자렬들을 비교한다. strcmp(s1,s2)함수는 두개의 문자렬 s1과 s2를 인수로 받아들이고 다음의 값을 돌려준다.(표 10-1)

표 10-1. strcmp()함수의 돌림값

값	조건
< 0	s1가 s2앞에 온다.
0	s1가 s2과 같다.
> 0	s1가 s2뒤에 온다.

문자렬들을 다음 식에 의해 호출한다.

```
(*ppl)->GetName();
```

인수 ppl은 지적자에로의 지적자이고 그것이 가리키는 지적자에 의해 지적된 이름을 요구한다. 성원호출연산자 ->는 1준위 간접참고하지만 다른 준위를 간접참고할 필요가 있으므로 ppl앞에 별표를 놓는다.

제 9 절. 구문해석실례

프로그람작성자는 기호문자렬을 해석하는 문제에 부닥치군 한다. 실례로 사용자가

건반으로 입력한 명령들, 자연언어로 된 문장들, 프로그람언어의 명령문, 대수식 같은 것을 들수 있다. 이제는 문자렬과 지적자를 배웠으므로 이러한 문제를 조종할수 있다.

이 장의 다음 실례는 6/3+2*3-1과 같은 산수식을 해석하는 방법을 보여준다. 사용자는 식을 입력하고 프로그람은 식의 문자를 하나씩 조사하면서 산수항의 의미를 찾고 결과값을 표시한다. 식에는 4개의 산수연산자 +, -, *, /를 사용한다. 프로그람을 쉽게 작성하기 위하여 수들을 한자리로 제한하고 괄호를 사용하지 않는다. 이 프로그람은 char형자료를 보관하도록 Stack클라스(실례 7-8)를 변경하였다. 두개의 수와 연산자를 보관하는데 탄창을 사용한다. 탄창은 LIFO(Last-In First-Out)용기이므로 식을 해석할 때 마지막으로 보관된 항목을 호출하는데 효과적인 기억기묶음이다. Stack클라스외에 Express라는 클라스를 사용하여 전체 산수식을 표시한다. Express클라스의 성원함수들은 사용자가 입력한 문자렬형식의 식으로 객체를 초기화하고 식을 해석하여 그 결과인 상수값을 돌려준다.

1. 산수식의 해석

여기에 산수식해석방법이 있다. 왼끝으로부터 시작하여 매개 문자를 차례로 검색한다. 문자는 수자('0'~'9')이거나 연산자 ('+', '-', '*', '/'문자)일수 있다.

문자가 수자이면 그것을 늘 탄창에 밀어넣는다. 또한 처음으로 만난 연산자도 밀어넣는다. 요점은 뒤에 오는 연산자들을 조종하는데 있다. 현재 연산자뒤에 오는 수자를 아직 읽지 못하여 그 연산자를 실행할수 없다고 하자. 연산자검색은 탄창에 보관된 이전 연산자를 실행할수 있다는 신호로 된다. 즉 탄창에 2+3이라는 렬이 있으면 더하기를 하기 전에 다른 연산자를 찾을 때까지 기다린다.

이리하여 현재 문자가 연산자(첫 연산자제외)라는것을 알게 되면 이전의 수(앞의 실례에서는 3)와 그 앞의 연산자(+)를 탄창에서 꺼내고 그것들을 변수 lastVal과 lastOp에 보관한다. 끝으로 첫째 수(2)를 꺼내고 두 수에 대한 산수연산을 수행한다.(즉 5를 얻는다.) 앞의 연산자를 항상 실행할수 있는것은 아니다.

*와 /는 +와 -보다 우선순위가 높으므로 식 3+4/2에서 나누기를 할 때까지 +를 실행할수 없다. 그러므로 식에서 /를 얻었을 때 나누기를 실행할 때까지 2와 +를 탄창에 밀어넣어야 한다.

다른 한편 현재연산자가 + 또는 -이면 앞연산자를 먼저 실행할수 있다. 즉 식 4-5+6에서 +를 발견하였지만 -를 실행하는것이 옳으며 6/2-3에서는 -를 발견하였지만 나누기를 하는것이 옳다. 표 10-2는 4가지 가능성을 보여준다.

표 10-2. 연산자와 구문해석동작

앞연산자	현재연산자	실례	동 작
+ 혹은 -	* 혹은 /	3+4/	앞연산자와 앞의 수(+,4)를 밀어넣는다.
* 혹은 /	* 혹은 /	9/3*	앞연산자를 실행하고 결과(3)을 밀어넣는다.

+ 혹은 -	+ 혹은 -	6+3+	앞연산자를 실행하고 결과(9)를 밀어넣는다.
* 혹은 /	+ 혹은 -	8/2-	앞연산자를 실행하고 결과(4)를 밀어넣는다.

Parse()성원함수는 입력식을 한 문자씩 더듬으면서 이 조작을 처리한다. 또한 다른 일감이 있다. 탄창은 여전히 한개 수 또는 수-연산자-수의 렬을 여러개 포함한다. 탄창을 내려가면서 이 렬을 실행할수 있다. 마지막 한개 수가 탄창우에 남는데 이것은 원시식의 값이다. Solve()성원함수가 이 일감을 수행하며 하나의 수가 남을 때까지만 탄창을 내려가면서 작업한다. 일반적으로 Parse()는 탄창우에 무엇인가 밀어넣으며 Solve()는 그것을 꺼낸다.

2. 구문해석프로그람

아래에 프로그람과의 대화가 있다.

```
2+3*4/3-2형식의 산수식을 입력하십시오.
수값은 한자리이여야 합니다.
공백이나 괄호는 허용되지 않습니다.
산수식을 입력하십시오: 9+6/3
결과는 5
계속하겠습니까(y/n)?
```

산수연산결과가 한자리이상의 수를 포함하면 옳다. 결과는 -128～127까지의 char 형의 수값크기에 의해서만 제한되고 입력문자렬은 0~9까지의 수로 제한된다.

(실례 10-25) 한자리수들로 구성된 산수식의 평가

```
#include <iostream>
#include <cstring>
using namespace std;
const int LEN = 80;
const int MAX = 40;
class Stack
{
private:
   char st[MAX];
   int top;
public:
   Stack() { top = 0; }
   void Push(char var) { st[++top] = var; }
   char Pop() { return st[top--]; }
   int GetTop() { return top; }
};
class Express
{
private:
   Stack s;
   char* pStr;
```

```
  int len;
public:
  Express(char* ptr) { pStr = ptr; len = strlen(pStr); }
  void Parse();
  int Solve();
};
void Express::Parse()
{
  char ch;
  char lastVal;
  char lastOp;
  for(int j=0; j<len; j++)
  {
     ch = pStr[j];
     if(ch >= '0' && ch <= '9')
         s.Push(ch - '0');
     else if(ch == '+' || ch == '-' || ch == '*' || ch == '/')
     {
         if(s.GetTop() == 1)
           s.Push(ch);
         else
         {
           lastVal = s.Pop();
           lastOp = s.Pop();
           if((ch == '*' || ch == '/') && (lastOp == '+' || lastOp == '-'))
           {
             s.Push(lastOp);
             s.Push(lastVal);
           }
           else
           {
             switch(lastOp)
             {
             case '+': s.Push(s.Pop() + lastVal); break;
             case '-': s.Push(s.Pop() - lastVal); break;
             case '*': s.Push(s.Pop() * lastVal); break;
             case '/': s.Push(s.Pop() / lastVal); break;
             default: cout << "\n알려지지 않은 연산입니다."; exit(1);
             }
           }
           s.Push(ch);
         }
     }
     else
     {
         cout << "\n알려지지 않은 입력문자";
         exit(1);
     }
```

```
    }
}
int Express::Solve()
 {
    char lastVal;
    while(s.GetTop() > 1)
    {
       lastVal = s.Pop();
       switch(s.Pop())
        {
       case '+': s.Push(s.Pop() + lastVal); break;
       case '-': s.Push(s.Pop() - lastVal); break;
       case '*': s.Push(s.Pop() * lastVal); break;
       case '/': s.Push(s.Pop() / lastVal); break;
       default: cout << "\n알려지지 않은 연산자입니다."; exit(1);
       }
    }
    return int(s.Pop());
}

int main()
 {
    char ans;
    char string[LEN];
    cout << "\n2+3*4/3-2형식의 산수식을 입력하십시오."
            "\n수값은 한자리이여야 합니다."
            "\n공백이나 괄호는 허용되지 않습니다.";
    do
    {
       cout << "\n산수식을 입력하십시오: "; cin >> string;
       Express* ePtr = new Express(string);
       ePtr->Parse();
       cout << "\n결과는 " << ePtr->Solve();
       delete ePtr;
       cout << "\n계속하겠습니까(y/n)? "; cin >> ans;
    } while(ans == 'y');
    return 0;
}
```

이것은 긴 프로그람이지만 이미 앞에서 설계한 클라스 Stack를 새로운 상황에서 사용하는 방법을 보여준다. 또한 지적자를 사용하는 여러가지 방법과 문자들의 배렬로서 문자렬을 다루는것이 얼마나 효과있는가를 보여준다.

3. 지적자오유수정

지적자는 리해할수 없는 치명적인 프로그람오유의 원천일수 있다. 가장 일반적인 문제는 프로그람작성자가 지적자변수에 유효주소를 보관하는데서 실패하여 지적자가

기억기의 어느 위치도 가리키지 않을 때 발생한다. 이때 지적자는 프로그람코드를 지적할수도 있고 혹은 조작체계를 가리킬수도 있다. 그때 프로그람작성자가 지적자를 사용하여 기억기에 값을 넣으면 그 값은 프로그람이나 조작체계를 변경시키게 되고 콤퓨터는 멎거나 다른 오동작을 일으킨다.

특히 이러한 현상은 지적자가 NULL이라는 주소 0을 지적할 때 발생한다. 실례로 지적자변수가 외부변수로 정의되면 외부변수는 자동적으로 0으로 초기화된다. 클라스의 실례변수들도 0으로 초기화된다. 여기에 그러한 경우를 보여주는 프로그람이 있다.

```
int* intptr;     // 기억변수 0으로 초기화한다.
void main()
{                       // intptr에 유효주소를 넣는데 실패.
   *intptr = 37;     // 0주소에 37을 넣으려고 시도.
}
```

intptr가 정의되면 그것이 외부변수이므로 값 0이 주어진다. 또한 하나의 프로그람 명령문에서 0주소에 값 37을 삽입하려고 한다.

그러나 프로그람을 실행하면 실시간오유검사부가 주소 0호출을 검출하여 오유통보(호출위반, null지적자대입, 페지중지)를 표시하고 프로그람도 완료한다. 그것은 지적자를 적당히 초기화하는데 실패했기때문이다.

요 약

우리는 콤퓨터기억기의 매개 바이트가 주소를 가지고 주소가 지적자상수라는것을 배웠다. 주소연산자 &를 사용하여 변수들의 주소를 검색할수 있다.

지적자는 주소값을 보관하는 변수이다. 지적자는 지적자를 표시하는 별표(*)를 사용하여 정의한다.

번역프로그람은 지적자가 무엇을 지적하는지 알아야 하므로 자료형이 늘 지적자정의에 포함된다.(void는 제외)

지적자에 대하여 정확히 산수연산을 할수 있다. 지적자가 가리키는 객체는 간접연산자 *를 사용하여 호출할수 있다. 간접연산자는 지적된 변수의 내용을 표시한다.

특수형void*는 임의의 형에로의 지적자를 의미한다. void*는 어떤 지적자가 다른 형의 주소를 보관해야 하는 경우에 사용한다. 배렬원소는 괄호를 가지는 배렬표기 또는 별표를 가지는 지적자표기를 사용하여 호출할수 있다. 다른 주소처럼 배렬의 주소도 상수이지만 변수에 대입하여 그 변수를 증가시키거나 감소시키는 방법으로 변경할수 있다.

변수의 주소를 함수에 넘길 때 함수는 원시변수와 작업할수 있다. (그러나 인수를 값에 의해 넘길 때에는 원시변수와 작업할수 없다.) 이때 지적자에 의한 넘기기는 지적자인수들이 비록 간접연산자에 의하여 비참고하거나 호출해야 하지만 참고에 의한

넘기기와 꼭같은 리득을 제공한다.

그러나 지적자는 일부 경우에 더 큰 융통성을 제공한다.

문자렬상수는 배렬 혹은 지적자로서 정의될수 있다. 지적자변수는 융통성이 있으나 지적자값이 못쓰게 될 위험이 있다. char형배렬인 문자렬은 일반적인 방법으로 함수에 넘기고 지적자를 사용하여 호출할수 있다.

new연산자는 체계로부터 지정된 량의 기억기를 얻어서 기억기에로의 지적자를 돌려준다. new연산자는 프로그람을 실행할 때 변수와 자료구조체를 창조하는데 쓰인다. delete연산자는 new로 얻은 기억기를 해방한다.

지적자가 객체를 가리킬 때 객체의 클라스성원은 성원호출연산자 ->에 의하여 호출할수 있다. 이러한 문법은 구조체성원호출에서 사용할수 있다.

클라스와 구조체는 자기 형에로의 지적자를 자료성원으로 포함할수 있다. 이것은 련결목록과 같은 복잡한 자료구조를 실현할수 있게 한다.

또한 지적자에로의 지적자가 있을수 있다. 이러한 변수는 int** pptr와 같이 두개의 별표를 사용하여 정의한다.

문 제

1. 변수 testVar의 주소를 표시하는 명령문을 쓰시오.

2. float형의 린접한 변수들을 지적하는 두개의 지적자의 내용은 4byte만큼 차이난다. 옳은가?

3. 지적자는

① 변수의 주소이다.

② 다음번에 호출하려는 변수에 대한 지시이다.

③ 주소를 보관하기 위한 변수이다.

④ 주소변수의 자료형이다.

어느것이 옳은가?

4. 다음것을 얻기 위한 식을 쓰시오.

① var의 주소

② var에 의해 지적된 변수의 내용

③ 참고인수로서 사용된 변수 var

④ 자료형char에로의 지적자

5. 주소와 지적자는 각각 무엇이라고 말할수 있는가?

6. float에서 지적자형의 변수정의를 쓰시오.

7. 지적자를 사용하는 하나의 방법은 무엇을 가지지 않는 기억주소를 참고하는것인가?

8. 지적자 testPtr가 변수 testVar를 가리킬 때 testVar의 내용을 이름을 사용하지 않고 표시하는 명령문을 쓰시오.

9. 자료형뒤에 배치된 별표는 무엇을 의미하며 변수이름앞에 배치된 별표는 무엇을 의미하는가?

10. 식 *test는

① test에로의 지적자

② test의 내용의 참고

③ test의 비참고

④ test에 의해 지적된 변수값에 대한 참고라고 말할수 있다.

어느것이 옳은가?

11. 다음의 코드가 옳은가?

```
int intVar = 333;
int* intPtr;
cout << *intPtr;
```

12. void에로의 지적자에는 어떤 형의 지적자를 보관할수 있는가?

13. intArr[3]과 *(intArr+3)의 차이는 무엇인가?

14. 지적자표기를 사용하여 배렬 intArr(77개 원소를 가진다.)의 매개 값을 표시하는 코드를 쓰시오.

15. intArr가 옹근수배렬일 때 식 intArr++는 왜 옳지 않은가?

16. 함수에로 인수를 넘기는 세가지 방법들중 어느 방법으로 함수가 호출측프로그람의 인수를 변경할수 있는가?

17. 지적자가 가리키는 변수의 형은 지적자의 정의부분이여야 한다. 그것은

① 산수연산을 위해서이다.

② 지적자들을 다른데 추가하여 구조체성원을 호출할수 있게 하기 위해서이다.

③ 번역프로그람이 배렬원소를 호출하여 산수연산을 정확히 진행하도록 하기 위해서이다.

어느것이 옳은가?

18. 지적자표기를 사용하여 Func()라는 함수의 선언을 쓰시오. 함수는 void형을 돌려주고 char형배렬을 인수로 가진다.

19. 지적자표기를 사용하여 문자렬 s1로부터 문자렬 s2에 80문자를 전송하는 코드를 쓰시오.

20. 문자렬의 첫 원소는

① 문자렬의 이름

② 문자렬의 첫 문자

③ 문자렬의 길이

④ 문자렬을 보관하는 배렬의 이름이다.

어느것이 옳은가?

21. 지적자표기를 사용하여 RevStr()라는 함수선언을 쓰시오. 함수는 문자렬값을 돌려주고 한개의 문자렬인수를 가진다.

22. 문자렬 "One", "Two", "Three"에로의 지적자들의 배렬 numPtrs의 정의를 쓰시오.

23. new연산자는

① 변수에로의 지적자를 돌려준다.

② new라고 부르는 변수를 창조한다.

③ 새 변수용의 기억기를 얻는다.

④ 얼마만한 기억기가 유효한가를 말한다.

어느것이 옳은가?

24. new를 사용하는것은 배렬을 사용하는것보다 적은 기억기를 소비한다. 옳은가?

25. delete연산자는 조작체계에 무엇을 돌려주는가?

26. UpperClass형의 객체를 가리키는 지적자 p가 주어졌다. 이 객체의 Exclu()성원함수를 실행하는 식을 쓰시오.

27. 배렬 objArr의 첨수 7인 객체가 주어졌을 때 이 객체의 성원함수 Exclu()를 실행하는 식을 쓰시오.

28. 련결목록에서

① 매개 련결은 다음 련결에로의 지적자를 포함한다.

② 지적자배렬은 련결들을 가리킨다.

③ 매개 련결은 자료 혹은 자료에로의 지적자를 포함한다.

④ 련결들은 배렬에 보관된다. 어느것이 옳은가?

29. float형의 변수들을 지적하는 8개 지적자들의 배렬 arr의 정의를 쓰시오.

30. 매우 큰 객체 혹은 구조체를 정렬한다면 다음과 같이 하는것이 효과있다.

① 그것들을 배렬에 보관하고 배렬을 정렬한다.

② 배렬에 그것들에로의 지적자들을 보관하고 배렬을 정렬한다.

③ 그것들을 련결목록에 보관하고 련결목록을 정렬한다.

④ 그것들에로의 참고를 배렬에 보관하고 배렬을 정렬한다.

어느것이 옳은가?

련습문제

1. 사용자로부터 한조의 수들을 읽어서 float형의 배렬에 보관하는 프로그람을 작성하시오. 배렬에 수들을 일단 보관한 다음 그 평균을 계산하고 결과를 출력하시오. 될수록 지적자표기를 사용하시오.

2. 실례 10-19의 String클라스에 문자렬을 모두 대문자로 변환하는 CopyIt()라는 성원함수를 추가하시오. 서고함수 toupper()를 사용하시오. toupper()는 한개 문자를 인수로 가지며 대문자로 변환된 문자를 돌려준다. 이 함수는 <cctype>머리부파일을 사용한다. main()에 CopyIt()를 시험하는 코드를 쓰시오.

3. 요일을 표시하는 문자렬에로의 지적자배렬(실례 10-17)을 사용하시오. BSort()와 Order()함수(실례 10-24)를 사용하여 문자렬들을 자모순으로 분류하는 프로그람을 쓰시오. 실제문자렬이 아니라 문자렬에로의 지적자를 보관하시오.

4. 실례 10-23에 해체자를 추가하시오. LinkList객체가 해체될 때 모든 련결을 삭제해야 한다. 사슬을 따라 더듬어나가면서 만나게 되는 모든 련결을 삭제할수 있다. 련결을 삭제할 때마다 통보문을 표시하여 해체자를 시험할수 있다.

5. main()에 세개의 float형국부배렬이 있다고 하자. 처음의 두개는 이미 초기화되여있다. 인수로서 세개의 배렬의 주소를 받아들여 처음 두개 배렬의 내용을 한 원소씩 더하여 결과를 셋째 배렬에 보관하여 돌려주는 AddArrays()함수를 추가하시오. 이 함수의 넷째 인수는 배렬의 크기이다. 지적자표기를 사용하시오. 배렬의 정의에만 괄호를 사용할수 있다.

6. 서고함수 strcmp(s1, s2)을 자체로 정의하시오. strcmp()는 두개의 문자렬을 비교하여 s1가 자모순으로 앞서면 -1, s1와 s2이 같으면 0, s2이 자모순으로 앞서면 1을 돌려준다. 자체의 함수 CompStr()를 정의하시오. 이 함수는 두개의 인수를 가지며 그것들을 한 문자씩 비교하고 int를 돌려준다. main()에서 CompStr()함수를 시험하시오. 그리고 철저히 지적자표기를 사용하시오.

7. 실례 10-24의 클라스를 변경하여 사람의 이름뿐아니라 생활비까지 표시하는 float형의 항목을 포함하시오. SetName()과 PrintName()성원함수들을 SetData()와 PrintData()로 변경하고 이 함수들에 이름은 물론 생활비를 설정하고 표시하는 능력까지 주어야 한다. 지적자표기를 사용하여 이름이 아니라 생활비에 의하여 persPtr배렬안의 지적자들을 정렬하는 SalSort()함수를 쓰시오. 실례 10-24와 같이 다른 함수를 호출하지 말고 SalSort()에서 직접 정렬하시오. 그러자면 ->가 *보다 우선순위가 높다는것을 잊지 말아야 한다. 말하자면

```
if((*(pp+j))->GetSalary()->(*(pp+k))->GetSalary())
{ … }
```

8. 실례 10-23에서 AddItem()성원함수를 변경하여 목록의 선두가 아니라 끝에 항목을 추가하도록 하시오. 이것은 처음에 삽입한 항목이 먼저 표시되게 한다. 즉

```
25
36
49
64
```

항목들을 추가하기 위해서는 목록의 끝까지 지적자사슬을 추적해야 한다.

9. 100개 옹근수를 보관하고 그것을 쉽게 호출하게 하려고 한다. 그런데 여기에 문제가 있다고 하자. 즉 콤퓨터안의 기억기가 단편화되여 사용가능한 최대배렬은 10개의 옹근수만 보유할수 있다. 이것은 매개 배렬이 10개의 옹근수를 가지는 int배렬을 10개 정의하고 이 배렬에로의 10개의 지적자로 이루어진 배렬을 정의하여 해결할수 있다. int배렬은 a0, a1, a2, …과 같은 이름을 가질수 있다. 이 배렬들의 매개 주소는 int*형의 지적자배렬에 보관한다. 그다음 ap[j][k](여기서 j는 ap안의 지적자걸음, k는 매개 배렬의 개별적인 옹근수)와 같은 식을 사용하여 개별적인 옹근수들을 호출할수 있다. 이것은 2차원배렬을 호출하는 느낌을 주지만 실제로는 1차원배렬들의 한 묶음이다. 이러한 배렬의 한 조에 시험용자료(0, 10, 20, …)를 채워넣고 그것이 정확한가를 자료를 표시하여 확인하시오.

10. 련습 9는 서로 다른 프로그람명령문에서 서로 다른 이름을 사용하여 10개의 int형배렬을 창조하기때문에 좀 불편하다. 또한 매개 주소는 개별적인 명령문에 의하여 얻어야 한다. new로 이것을 단순화할수 있다. 배렬들을 순환하면서 new에 의하여 그것들에로의 지적자를 대입한다.

```
for(j=0; j < NUMARRAYS; j++)
  *(ap+j) = new int[MAXSIZES];
```

이 방법을 사용하여 문제 9의 프로그람을 다시 작성하시오. 문제 9와 같은 식을 사용하여, 또는 지적자표기 *(*(ap + j) + k)를 사용하여 개별적배렬의 원소를 호출할수 있다. 이 두개 식은 등가하다.

11. 유일한 변수를 가지는 배렬표기를 사용하여 하나의 1차원배렬로 문제 10과 같이 10개의 개별적배렬을 취급하도록 클라스를 창조하시오. 즉 main()의 명령문들은 a[j]와 같은 식에 의하여 원소들을 호출할수 있다. 클라스의 성원함수들은 두개의 걸음으로 자료를 호출해야 한다. 그 결과를 얻을수 있도록 첨수연산자를 재정의하시오. 배렬에 시험자료를 채워넣고 그것을 표시하시오. main()에서 원소들을 호출하는데 클라스대면부의 배렬표기를 사용하지만 클라스성원함수의 실현에서는 지적자표기만 사용해야 한다.

12. 지적자는 복잡히 얽혀져있으므로 그 조작을 클라스로 모의하여 쉽게 리해할수 있다. 지적자조작을 밝히기 위하여 배렬로서 콤퓨터기억기를 모의한다. 이 방법은 아주 리해하기 쉬우므로 지적자를 사용하여 기억기를 호출할 때 실제로 무엇을 하려고 하는가를 알수 있다. char형의 유일한 배렬을 사용하여 변수의 모든 형을 보관한다. 이것은 콤퓨터기억기가 실제로 무엇인가를 보여준다. 즉 바이트(char형과 같은 크기)배렬에서 매개 바이트는 주소(배렬의 첨수)를 가진다. 그러나 일반적으로 C++는 char형배렬에 float 또는 int를 보관할수 없다. 그러므로 보관하려는 매개 자료형에 대하여 개별적배렬로서 기억기를 모의한다. 이 문제에서는 어떤 수값들을 float로 고찰하고 그 형의 배렬을 요구한다. 그것을 fmemory라고 하자. 그러나 지적자값(주소)도 기억기에

보관되므로 그것들을 보관하는데 다른 배렬이 요구된다. 주소모의에 배렬첨수를 사용하고 그 첨수의 최대배렬을 int형으로 보관하므로 int형의 배렬을 창조하고 그것을 pmemory라고 한다. 그리고 여기에 지적자를 보관한다.

fmemTop라는 fmemory에로의 첨수는 float값을 보관할수 있는 다음의 유효위치를 가리킨다. pmemory에 류사한 첨수 pmemTop가 있다. 기억기를 벗어나는데 대해서는 적당한 방도가 없다. 이 배렬들이 충분히 커서 무엇인가 보관할 때마다 배렬의 다른 첨수에 그것을 간단히 삽입할수 있다. 기억관리도 걱정하지 마시오.

Float라는 클라스를 창조하시오. 실제기억기대신에 fmemory에 보관하고 float형의 수들을 모의하는데 이 클라스를 사용하시오. Float의 유일한 실제자료는 자체의 주소 즉 float가 fmemory안에 보관되는 첨수이다. 그 실례를 addr라고 하시오. 또한 클라스 Float는 두개의 성원함수를 가진다. 첫째 함수는 1인수구성자로서 Float를 float값으로 초기화한다. 이 구성자는 fmemTop가 가리키는 fmemory의 원소에 float값을 보관하며 addr에 fmemTtop의 값을 보관한다. 이것은 번역 및 련결프로그람이 실제기억기안에 일반변수를 배치하는 방법과 비슷하다. 둘째 성원함수는 재정의된 &연산자이다. 이것은 단순히 지적자(실제로 형첨수)값을 돌려준다.

PtrFloat라는 둘째 클라스를 창조하시오. 이 클라스의 실제자료는 다른 주소(변수)와 보관하는 pmemory안의 주소(변수)를 보관한다. 성원함수는 이 지적자를 int첨수값으로 초기화한다. 둘째 성원함수는 재정의된 간접연산자 *이다. 간접연산자는 pmemory로부터 그 자료(주소이기도 하다.)와 보관된 주소를 얻는다. 그다음 새로운 주소를 fmemory에로의 첨수로 사용하여 그 주소자료에 의해 지적된 float값을 얻는다.

```
Float& PtrFloat::operator*()
{
  return fmemory[pmemory[addr]];
}
```

이러한 방법으로 간접연산자 *의 동작을 모의한다. 갈기기호의 왼변에 *를 사용할수 있도록 이 함수로부터 참고에 의해 귀환하여야 한다.

두개의 클라스 Float와 PtrFloat가 비슷하지만 Float는 기억기를 표시하는 배렬에 float를 보관하고 PtrFloat도 역시 기억기를 표시하는 다른 배렬에 int(기억지적자 그러나 실제로 배렬첨수값)를 보관한다.

이 클라스의 사용실례가 있다.

```
Float var1 = 1.234;
Float var2 = 5.678;
PtrFloat ptr1 = &var1;
PtrFloat ptr2 = &var2;
cout << "*ptr1=" << *ptr1;
cout << "*ptr2=" << *ptr2;
*ptr1 = 7.123;
*ptr2 = 7.456;
```

```
cout << "*ptr1=" << *ptr1;
cout << "*ptr2=" << *ptr2;
```

아래에 프로그람의 출력이 있다.

```
*ptr1 = 1.234
*ptr2 = 5.678
*ptr1 = 7.123;
*ptr2 = 7.456
```

이것은 지적자를 실현하는 간접적인 방법 같아보이지만 지적자와 주소연산자의 내부동작을 공개함으로써 그 본질에 대한 다른 견해를 제공한다.

제 11 장. 가상함수

이제는 지적자를 리해하였으므로 더 고급한 C++개념을 학습할수 있다.

이 장에서는 가상함수, 동료함수, 정적함수, 재정의된 대입연산자, 재정의된 복사구성자, this지적자, typeid에 의한 객체의 클라스검색에 대하여 설명한다. 이것들은 고급한 개념들이다. 이것은 모든 C++프로그람에 요구되지 않지만 여러 프로그람들에서 널리 쓰이고있다. 특히 가상함수는 다형성의 본질이다. 다형성은 객체지향프로그람작성법의 중요한 특성의 하나이다.

제 1 절. 가상함수

가상(virtual)이라는것은 표현상으로 존재하고 실제로는 존재하지 않는것이다. 가상함수를 사용할 때 어떤 클라스의 함수를 호출하는것처럼 보이는 프로그람이 실제로는 다른 클라스의 함수를 호출한다.

그러면 가상함수가 왜 요구되는가?

그것은 각이한 클라스의 많은 객체가 있는데 그것들을 모두 하나의 배렬에 넣어서 같은 함수호출에 의하여 특수한 조작을 수행하기 위해서이다. 실례로 다른 모양을 가지는 도형 즉 3각형, 원, 4각형 등을 포함하는 도형을 그리는 프로그람을 고찰하자.

도형의 매개 클라스에는 객체를 화면에 그리는 Draw()성원함수가 있다. 이 원소들을 모두 묶어서 하나의 그림을 그릴 계획이라고 하자. 그리고 관례적인 수법으로 그림을 그리려고 한다. 한가지 방법은 그림의 매개 객체에로의 지적자들을 보관하는 배렬을 하나 창조하는것이다. 배렬은 다음과 같이 정의할수 있다.

```
Shape*  ptrArr[100]; // 도형에로의 지적자 100개의 배렬
```

이 배렬에 모든 도형들에로의 지적자를 보관한다면 전체 그림은 간단히 순환을 사용하여 그릴수 있다.

```
for(int j=0; j<N; j++)
   ptrArr[j]->Draw();
```

이것은 놀랄만한 능력이다. 즉 전혀 다른 함수들이 같은 함수호출에 의해 실행된다. ptrArr안의 지적자가 원을 지적하면 원을 그리는 함수가 호출되고 지적자가 3각형을 지적하면 3각형을 그리는 함수가 호출된다. 이것이 다형성이다. 함수들은 같은 식 즉 Draw()를 가지지만 ptrArr[j]의 내용에 따라서 실제로는 다른 함수들이 호출된다. 다형성은 클라스의 계승과 함께 객체지향프로그람작성법의 중요한 특성의 하나이다.

다형적인 환경에서 작업하려면 두가지 조건을 만족시켜야 한다. 우선 원과 3각형과 같이 서로 다른 클라스의 모든 도형을 하나의 기초클라스에서 파생시켜야 한다. 다음으로 기초클라스에서 Draw()함수를 가상으로 선언하여야 한다.

1. 지적자를 사용하여 호출하는 일반성원함수

실례 11-1은 기초클라스와 파생클라스들에 이름이 같은 일반성원함수(비가상함수)를 포함하고 지적자를 사용하여 이 함수를 호출할 때 어떻게 동작하는가를 보여준다.

(실례 11-1) 지적자로부터 호출되는 일반성원함수

```
#include <iostream>
using namespace std;
class Base
{
public:
    void Show() { cout << "Base\n"; }
};
class Derv1 : public Base
{
public:
    void Show() { cout << "Derv1\n"; }
};
class Derv2 : public Base
{
public:
    void Show() { cout << "Derv2\n"; }
};
int main()
{
    Derv1 dv1;
    Derv2 dv2;
    Base* ptr;
    ptr = &dv1;
    ptr->Show();
    ptr = &dv2;
    ptr->Show();
    return 0;
}
```

클라스 Derv1과 Derv2는 클라스 Base에서 파생되고 매개 클라스는 Show()라는 성원함수를 가진다. main()에서는 클라스 Derv1과 Derv2의 객체들과 클라스 Base에로의 지적자를 창조한 다음 기초클라스지적자에 파생클라스객체의 주소를 대입한다. 즉

```
ptr = &dv1;
```

어떤 형(Derv1)의 주소를 다른 형(Base)의 지적자에 대입할 때 번역프로그람이 왜 오유를 통보하지 않는가?

번역프로그람이 형검사를 할 때 사용하는 규칙의 하나로서 파생클라스의 객체들에로의 지적자가 기초클라스의 객체들과 형호환된다는것이 있다.

다음 행 ptr->Show();을 실행할 때 어느 함수를 호출하는가?

그것은 Base::Show()인가, Derv1::Show()인가. 또한 실례 11-1의 마지막 둘째 행

에서 지적자에 클라스 Derv2의 객체의 주소를 대입하고 다시 ptr->Show();를 실행한다.

여기서는 어느 Show()함수가 호출되는가?

프로그람의 출력이 여기에 대답을 준다.

```
Base
Base
```

이와 같이 기초클라스의 함수가 항상 실행된다. 번역프로그람은 지적자 ptr의 내용을 무시하고 그림 11-1과 같이 지적자형과 일치하는 성원함수를 선택한다.

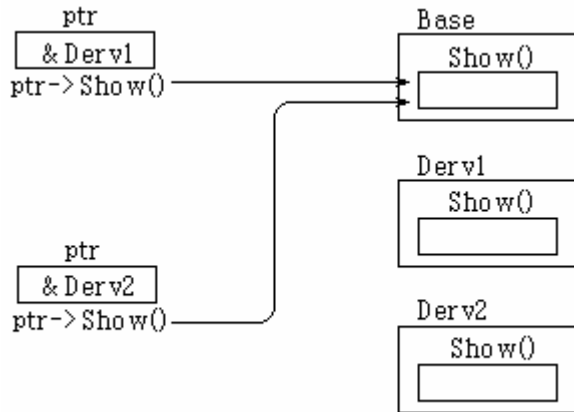



그림 11-1. 비가상적인 지적자호출

이것은 이 절의 앞에서 제기한 문제를 해결하지 못한다. 즉 같은 명령문을 사용하여 다른 클라스의 객체들을 호출하지 못한다.

2. 지적자를 사용하여 호출하는 가상성원함수

여기서 프로그람을 조금 변경하자. 즉 기초클라스의 Show()함수앞에 virtual예약어를 쓴다.

(실례 11-2) 지적자로부터 호출되는 가상함수

```
#include <iostream>
using namespace std;
class Base
{
public:
   virtual void Show() { cout << "Base\n"; }
};
class Derv1 : public Base
{
public:
   void Show() { cout << "Derv1\n"; }
};
class Derv2 : public Base
```

```
{
public:
  void Show() { cout << "Derv2\n"; }
};
int main()
{
  Derv1 dv1;
  Derv2 dv2;
  Base* ptr;
  ptr = &dv1;
  ptr->Show();
  ptr = &dv2;
  ptr->Show();
  return 0;
}
```

이때 프로그람의 출력은 다음과 같다.

```
Derv1
Derv2
```

이와 같이 기초클라스가 아니라 파생클라스의 성원함수가 실행된다. ptr의 내용을 Derv1의 주소로부터 Derv2의 주소로 변경하여 Derv2클라스의 Show()를 실행할수 있다. 즉 같은 함수호출 ptr->Show();은 ptr의 내용에 따라서 다른 함수를 실행한다. 번역프로그람은 실례 11-1과 같이 지적자형에 따라서가 아니라 지적자 ptr의 내용에 따라서 함수를 선택한다. 이것을 그림 11-2에서 보여준다.

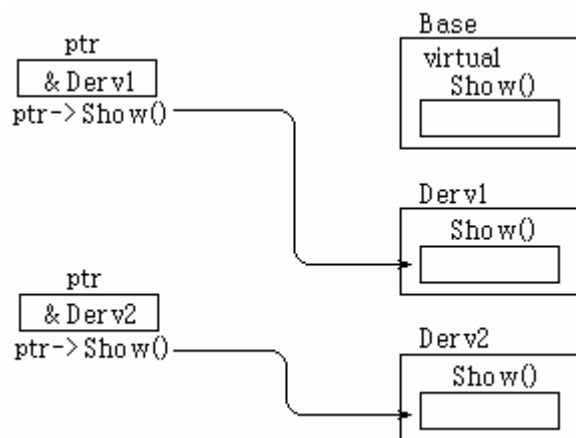



그림 11-2. 가상지적자호출

3. 후(동적)속박

번역프로그람이 어느 함수를 번역하는가 걱정할수 있다. 실례 11-1의 식 ptr->Show();은 항상 기초클라스의 Show()함수를 호출한다. 그러나 실례 11-2에서 번역프로그람은 ptr의 내용을 포함하는 클라스가 무엇인지 모른다. 그것은 클라스 Derv1 또는 Derv2의 객체의 주소이다.

그러면 번역프로그람은 어느 클라스의 Draw()를 호출하는가?

사실상 번역프로그람은 프로그람을 실행하기전까지 어떻게 해야 할지 모른다. 실행시에 ptr가 어느 클라스를 지적하는가를 알았을 때 그 클라스의 Draw를 호출한다.

이것을 후속박(late binding) 또는 동적속박(dynamic binding)이라고 한다.

이와 대조적으로 번역시에 일반적인 방법으로 함수를 선택하는것을 초속박(early binding) 또는 정적속박(static binding)이라고 한다.

동적속박은 추가적인 내용을 요구하지만 능력과 융통성을 제고한다.

4. 추상클라스와 순수가상함수

실례 9-6의 Shape클라스를 고찰하자.

Shape 클라스의 객체는 만들수 없지만 원, 4각형과 같은 도형은 만들수 있다. 이와 같이 기초클라스의 실례객체가 요구되지 않을 때 그것을 추상(abstract)클라스라고 한다. 추상클라스는 객체의 실례를 만드는데 쓰이는 파생클라스들의 어미로서 사용하기 위하여서만 존재한다. 또한 추상클라스는 클라스계층의 대면부를 제공해준다.

그러면 기초클라스의 객체실례를 만들 필요가 없는 클라스를 쓰는데서 무엇을 고려해야 하는가?

이것을 문서화해넣고 사용자들이 기억하게 할수 있으나 그러한 실례를 창조하지 못하도록 클라스를 쓰는것이 더 좋다. 그것을 실현하는 방법은 기초클라스에 적어도 하나의 순수가상함수를 배치하는것이다. 순수가상함수(pure virtual function)는 선언에 식 =0을 붙인것이다. 이것을 실례 11-3에세 보여준다.

(실례 11-3) 순수가상함수

```
#include <iostream>
using namespace std;
class Base
{
public:
   virtual void Show() = 0;
};
class Derv1 : public Base
{
public:
   void Show() { cout << "Derv1\n"; }
};
class Derv2 : public Base
{
public:
   void Show() { cout << "Derv2\n"; }
};
int main()
{
```

```
//Base bad; // 오유: 추상클라스의 객체를 만들수 없다.
   Base* arr[12];
   Derv1 dv1;
   Derv2 dv2;
   arr[0] = &dv1;
   arr[1] = &dv2;
   arr[0]->Show();
   arr[1]->Show();
   return 0;
}
```

여기서는 가상함수 Show()를 다음과 같이 선언한다.

```
virtual void show()=0;
```

여기서 같기기호는 대입을 의미하지 않으며 값 0은 어디에도 대입되지 않는다. =0 문법은 함수가 순수가상함수라는것을 번역프로그람에 알리는 방법을 보여준다. 이때 main()에서 클라스 Base의 객체들을 창조하려고 하면 번역프로그람은 추상클라스의 객체실례를 만들려고 한다고 오유를 통보한다. 또한 추상클라스로 만드는 순수가상함수의 이름을 통보한다. 이것이 유일한 선언이라고 하더라도 기초클라스의 Show()의 정의를 쓸 필요는 없다.

일단 기초클라스에 순수가상함수를 배치하면 객체의 실례를 만들려는 모든 파생클라스에서 그것을 재정의(overriding)해야 한다.

어떤 파생클라스에서 순수가상함수를 재정의하지 않으면 그 클라스도 추상클라스로 되므로 객체를 창조할수 없다.

일관성을 담보하기 위하여 기초클라스의 가상함수들을 모두 순수가상함수로 할수 있다. 실례 11-3에서는 객체들의 주소를 지적자배렬에 보관하고 배렬원소를 사용하여 그 성원함수를 호출한다. 이것은 유일한 지적자를 사용할 때처럼 동작한다. 실례 11-3의 출력은 다음과 같다.

```
Derv1
Derv2
```

5. 가상함수와 Person클라스

이제는 가상함수기구를 리해하였으므로 그 사용법을 고찰하자.

다음 실례는 실례 10-22와 실례 10-24를 확장한것이다. 모두 Person클라스를 사용하지만 두개의 파생클라스 Student와 Professor를 추가한다. 매개 클라스는 각각 IsOutstanding()이라는 함수를 포함한다. IsOutstanding()함수는 대학의 관리일군들이 실력있는 대학생들과 교수들의 목록을 쉽게 창조하게 한다.

(실례 11-4) Person클라스에서 가상함수

```
#include <iostream>
using namespace std;
class Person
```

```
{
protected:
  char name[40];
public:
  void GetName()
  {
     cout << "\n이름을 입력하십시오:";
     cin >> name;
  }
  void PutName()
     { cout << "\n 이름=" << name; }
  virtual void GetData() = 0;
  virtual bool IsOutStanding() = 0;
};
class Student : public Person
{
private:
  float gpa; // 평균성적
public:
  void GetData()
  {
     Person::GetName();
     cout << "평균성적을 입력하십시오:";
     cin >> gpa;
  }
  bool IsOutStanding() { return (gpa > 3.5) ? true : false; }
};
class Professor : public Person
{
private:
  int numPubs;
public:
  void GetData()
  {
     Person::GetName();
     cout << "출판물 건수를 입력하십시오:";
     cin >> numPubs;
  }
  bool IsOutStanding() { return (numPubs > 100) ? true : false; }
};
int main()
{
  Person* persPtr[100];
  int n = 0;
  char choice;
  do
  {
     cout << "대학생(s) 혹은 교수(p)를 입력하십시오: ";
```

```
        cin >> choice;
        if(choice == 's')
            persPtr[n] = new Student;
        else
            persPtr[n] = new Professor;
        persPtr[n++]->GetData();
        cout << "계속하겠습니까(y/n)? ";
        cin >> choice;
    } while(choice == 'y');
    for(int j=0; j<n; j++)
    {
        persPtr[j]->PutName();
        if(persPtr[j]->IsOutStanding())
            cout << "  이 사람은 우수한 사람입니다.\n";
    }
    return 0;
}
```

1) 클라스

Person클라스는 순수가상함수 GetData()와 IsOutstanding()을 포함하므로 추상클라스이고 객체를 창조할수 없다. Person클라스는 Student와 Professor클라스의 기초클라스로서 존재한다. 클라스 Student와 Professor는 기초클라스에 새로운 자료항목들을 추가한다. Student클라스는 학생의 학기평균성적을 표시하는 float형변수 gpa를 포함한다. Professor클라스는 교수가 출판물에 낸 출판물 건수를 표시하는 int형변수 numPubs를 포함한다. gpa가 3.5이상인 대학생들과 numPubs가 100건이상인 교수는 최우수자로서 고찰된다.

2) IsOutstanding()함수

IsOutstanding()함수는 Person에서 순수가상함수로 선언된다. Student클라스에서 이 함수는 gpa가 3.5이상이면 bool형값 true, 그렇지 않으면 false를 돌려준다. Professor에서는 교수의 numPubs변수가 100이상이면 true를 돌려준다. GetData()함수는 사용자에게 대학생의 gpa를 물으며 교수일 때는 출판물 건수를 묻는다.

3) main()프로그람

main()에서는 우선 사용자에게 여러명의 대학생과 교수의 이름을 입력하게 한다. 대학생의 경우에는 gpa를 묻고 교수의 경우에는 출판물 건수를 묻는다. 사용자가 입력을 끝내면 대학생과 교수들 모두의 이름을 출력하고 그들이 우수한 사람인가를 출력한다. 여기에 대화가 있다.

```
대학생(s) 혹은 교수(p)를 입력하십시오: s
이름을 입력하십시오: 김철수
평균성적을 입력하십시오: 3
계속하겠습니까(y/n)? y
대학생(s) 혹은 교수(p)를 입력하십시오: s
이름을 입력하십시오: 조순옥
```

```
평균성적을 입력하십시오: 4.5
계속하겠습니까(y/n)? y
대학생(s) 혹은 교수(p)를 입력하십시오: s
이름을 입력하십시오: 최철영
평균성적을 입력하십시오: 3.2
계속하겠습니까(y/n)? y
대학생(s) 혹은 교수(p)를 입력하십시오: p
이름을 입력하십시오:리준호
출판물 건수를 입력하십시오:112
계속하겠습니까(y/n)? y
대학생(s) 혹은 교수(p)를 입력하십시오: p
이름을 입력하십시오: 박철
출판물건수를 입력하십시오:79
계속하겠습니까(y/n)? n
이름=김철수
이름=조순옥
  이 사람은 우수한 사람입니다.
이름=최철영
이름=리준호
  이 사람은 우수한 사람입니다.
이름=김철수
```

6. 도형실례에서 가상함수

가상함수의 다른 실례로서 실례 9-6으로부터 파생된 도형실례를 고찰하자. 이 절의 앞에서 언급한것처럼 같은 명령을 사용하여 많은 도형을 그리거나 면적을 계산할수 있다. 실례 11-5는 이것을 수행한다.

(실례 11-5) Shape에서 가상함수의 사용

```
#include <iostream>
using namespace std;
class Shape
{
protected:
  int xCo, yCo;
public:
  Shape() : xCo(0), yCo(0) {}
  Shape(int x, int y) : xCo(x), yCo(y) {}
  virtual void ShowArea() const { cout << "면적="; }
};
class Ball : public Shape
{
private:
  int radius; // (xCo, yCo)는 중심
public:
  Ball() : Shape(), radius(0) {}
  Ball(int x, int y, int r) : Shape(x, y), radius(r) {}
  void ShowArea() const
```

```
    {
        Shape::ShowArea();
        cout << 3.141592 * radius * radius << endl;
    }
};
class Rect : public Shape
{
private:
    int width, height;
public:
    Rect() : Shape(), height(0), width(0) {}
    Rect(int x, int y, int h, int w) : Shape(x, y), height(h), width(w) {}
    void ShowArea() const
    {
        Shape::ShowArea();
        cout << height * width << endl;
    }
};
class Tri : public Shape
{
private:
    int width, height;
public:
    Tri() : Shape(), height(0), width(0) {}
    Tri(int x, int y, int h, int w) : Shape(x, y), height(h), width(w) {}
    void ShowArea() const
    {
        Shape::ShowArea();
        cout << height * width / 2.0 << endl;
    }
};
int main()
{
    int j;
    Shape* pShapes[3];
    pShapes[0] = new Ball(40, 12, 5);
    pShapes[1] = new Rect(12, 7, 10, 15);
    pShapes[2] = new Tri(60, 7, 12, 8);
    for(j=0; j<3; j++)
        pShapes[j]->ShowArea();
    for(j=0; j<3; j++)
        delete pShapes[j];
    return 0;
}
```

실례 11-5의 클라스지적자는 실례 9-6과 비슷하지만 Shape클라스의 ShowArea()함수가 순수가상함수라는것이 다르다.

main()에서는 배렬 ptrArr를 도형들의 지적자들로 설정한다. 다음으로 매개 클라스

의 세개의 객체를 창조하고 배렬에 그 주소를 대입한다. 그러면 세개 도형의 면적을 모두 계산하기 쉽다. 명령문 ptrArr[j]->ShowArea();은 순환변수가 변경될 때마다 면적을 표시한다.

이것은 대량적인 객체들을 하나의 단위로 묶어서 처리할 필요가 있을 때 특별히 강력한 방법으로 된다.

7. 가상해체자

기초클라스의 해체자는 항상 가상이여야 한다. 파생클라스객체를 해체하기 위하여 파생클라스객체에로의 기초클라스지적자를 delete와 함께 사용한다. 기초클라스해체자가 가상이 아니면 일반성원함수처럼 delete는 파생클라스의 해체자가 아니라 기초클라스의 해체자를 호출한다. 이것은 객체의 기초부분만 해체하게 한다. 실례 11-6은 이것을 보여준다.

(실례 11-6) 비가상 및 가상해체자

```
#include <iostream>
using namespace std;
class Base
{
public:
   ~Base()     // 비가상해체자
      { cout << "Base를 해체하였습니다.\n"; }
// virtual ~Base() { cout << "Base를 해체하였습니다.\n"; }
};
class Derv : public Base
{
public:
   ~Derv() { cout << "Derv를 해체하였습니다.\n"; }
};
int main()
{
   Base* pBase = new Derv;
   delete pBase;
   return 0;
}
```

프로그람의 출력은 다음과 같다.

```
Base를 해체하였습니다.
```

이것은 객체의 Derv부분의 해체자가 호출되지 않았다는것을 보여준다. 프로그람에서 기초클라스의 해체자는 가상함수가 아니지만 이 부분을 설명문으로 하고 두번째 가상함수의 설명부분을 실제로 명령문으로 하면 다음과 같이 출력한다.

```
Derv를 해체하였습니다.
Base를 해체하였습니다.
```

그러면 파생클라스객체의 두 부분이 적당히 해체된다. 물론 여기서 해체자가 특별

히 하는 일은 없으므로 가상해체자는 중요하지 않다. 그러나 일반적으로 파생클라스의 객체가 적당히 해체되도록 담보하기 위하여 모든 기초클라스의 해체자를 가상으로 해야 한다.

8. 가상기초클라스

다중계승과 관련한 가상기초클라스를 고찰하자.

기초클라스 Parent와 그 파생클라스인 Child1과 Child2 그리고 Child1과 Child2로부터 파생된 네번째 클라스 GrandChild로 되여있는 그림 11-3을 고찰하자.

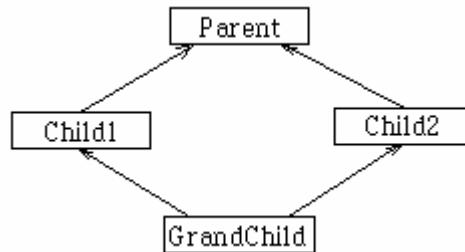



그림 11-3. 가상기초클라스

이 경우에 GrandChild클라스의 성원함수가 Parent클라스안의 자료나 함수를 호출할 때 문제가 생긴다. 실례 11-7에서 그것을 보여준다.

(실례 11-7) 기초클라스의 모호한 완료

```
#include <iostream>
using namespace std;
class Parent
{
protected:
   int baseData;
};
class Child1 : public Parent { };
class Child2 : public Parent { };
class GroundChild : public Child1, public Child2
{
public:
   int GetData()
      { return baseData; }  // 오유: 모호하다
};
```

GrandChild의 GetData()성원함수가 Parent의 baseData를 호출하면 번역프로그람 오유가 발생한다. Child1과 Child2가 Parent에서 파생될 때 매개 파생클라스는 Parent의 사본을 계승한다. 이 사본을 보조객체(subobject)이라고 한다.

그러면 GrandChild가 baseData를 참고할 때 두개의 사본들중 어느것을 호출하겠는가?

이 경우에 모호하므로 모호성을 제거하기 위하여 Child1과 Child2를 가상기초클라

스로 만든다.

(실례 11-8) 가상기초클라스

```
#include <iostream>
using namespace std;
class Parent
{
protected:
  int baseData;
};
class Child1 : virtual public Parent { };
class Child2 : virtual public Parent { };
class GroundChild : public Child1, public Child2
{
public:
  int GetData()
    { return baseData; }  // 오유: 모호하다.
};
```

이 두 클라스에서 예약어 virtual은 기초클라스 Parent의 단일한 공통보조객체를 공유하게 한다. baseData의 유일한 사본이 있으므로 GrandChild를 참고할 때 모호성은 없다.

제 2 절. 동료함수

밀봉성과 자료은페의 개념은 비성원함수들이 객체의 비공개 혹은 보호자료를 호출하지 못하게 한다. 따라서 성원함수가 없는 경우에는 그 자료를 얻을수 없다. 그러나 밀봉성이 아주 불리한 경우가 있다.

1. 다리로서의 동료

함수가 두개의 다른 클라스에 대하여 조작하게 하려는 경우에 두개 클라스의 객체들을 인수로 가지게 하고 그것들의 비공개자료에 대하여 조작한다. 이러한 경우에 동료함수(friend function)를 사용한다. 실례 11-9는 동료함수가 두 클라스를 이어주는 다리의 기능을 수행한다는것을 보여준다.

(실례 11-9) 동료함수

```
#include <iostream>
using namespace std;
class Beta;
class Alpha
{
private:
  int data;
public:
```

```
    Alpha() : data(3) {}
    friend int FriFunc(Alpha, Beta);
};
class Beta
{
private:
    int data;
public:
    Beta() : data(7) {}
    friend int FriFunc(Alpha, Beta);
};
int FriFunc(Alpha a, Beta b)
{
    return (a.data + b.data);
}
int main()
{
    Alpha aa;
    Beta bb;
    cout << FriFunc(aa, bb) << endl;
    return 0;
}
```

프로그람에는 두개의 클라스 Alpha와 Beta가 있다. 클라스들의 구성자는 단일자료항목들을 고정값으로 초기화한다. (Alpha에서 3, Beta에서 7)

함수 FriFunc()가 두개 클라스의 비공개자료성원들을 호출할수 있게 그것을 동료함수로 만든다. FriFunc()함수는 두 클라스에 frind예약어와 함께 삽입된다.

```
friend int FriFunc(Alpha, Beta);
```

이 선언은 클라스의 어디에나 배치할수 있으며 그것이 public부분인가, private부분인가에 관계없다. 매개 클라스의 객체는 함수 FriFunc()에 인수로 넘어가고 함수는 인수들을 통하여 두 클라스의 비공개자료성원들을 호출한다. FriFunc()함수는 단순히 자료항목들을 더하여 그 합을 돌려준다. main()프로그람은 이 함수를 호출하고 결과를 출력한다.

클라스는 그것을 선언할 때까지 참고할수 없다. 클라스 Beta를 클라스 Alpha의 FriFunc()함수의 선언에서 참고하므로 Alpha의 앞에서 선언하여야 한다.

이로부터 선언

```
class Beta;
```

를 프로그람의 선두에 놓는다.

2. 동료함수의 론의

동료함수를 론의할 필요가 있다. 동료함수는 언어에 융통성을 추가하는 한편 오직 성원함수들만 클라스의 비공개자료를 호출하게 하는 자료은폐를 유지하지 못하게 한다.

그러면 동료함수를 사용할 때 자료의 완전성은 어떠한가?

동료함수는 그 자료를 호출하는 클라스안에서 선언되여야 한다.

이처럼 클라스의 원천코드에로 호출하지 못하는 프로그람작성자는 어떤 함수를 동료로 만들수 없다. 이 점에서 클라스의 완전성은 여전히 보호된다. 하지만 동료함수는 개념적으로 불결하고 많은 동료들이 클라스들사이의 명백한 한계를 흐리게 한다. 그러므로 동료함수는 될수록 적게 사용해야 한다.

friend를 사용하는 프로그람실례를 고찰하자.

3. 거리실례

동료함수의 가장 일반적인 실례는 재정의된 연산자들의 만능성을 증가시키는데 friend를 사용하는 경우이다. 실례 11-10은 friend를 사용하지 않을 때 연산자사용에서의 제한을 보여준다. 이 실례는 실례 8-5와 8-13의 프로그람들을 변경한것이다.

(실례 11-10) +연산자재정의의 제한

```
#include <iostream>
using namespace std;
class Distance
{
private:
   int meters;
   float centies;
public:
   Distance() : meters(0), centies(0.0) {}
   Distance(float fltMeters)
   {
     meters = static_cast<int>(fltMeters);
     centies = 100 * (fltMeters - meters);
   }
   Distance(int me, float ce) { meters = me; centies = ce; }
   void ShowDist() { cout << meters << "m " << centies << "cm"; }
   Distance operator+(Distance);
};
Distance Distance::operator+(Distance d2)
{
   int m = meters + d2.meters;
   float c = centies + d2.centies;
   if(c >= 100.0) { c -= 100.0; m++; }
      return Distance(m, c);
}
int main()
{
   Distance d1 = 2.5, d2 = 1.25;
   Distance d3;
   cout << "\nd1="; d1.ShowDist();
```

```
    cout << "\nd2="; d2.ShowDist();
    d3 = d1 + 10.0;    // 옳다: Distance + float
    cout << "\nd3="; d3.ShowDist();
 // d3 =10.0 + d1;// 오유: float + Distance
 // cout << "\nd3=";
 // d3.ShowDist();
    cout << endl;
    return 0;
 }
```

프로그람에서 +연산자는 Distance형의 두개의 객체를 더할수 있게 재정의된다. 또한 메터와 센치메터의 10진소수를 나타내는 float형값을 Distance값으로 변환하는 1인수구성자가 있다.(즉 10.25를 10m 25cm로 변환한다.)

이러한 구성자가 존재할 때 main()에 다음과 같은 명령문을 쓸수 있다.

```
d3 = d1 + 10.0;
```

재정의된 +연산자는 오른변과 왼변에 모두 Distance형의 객체를 가진다. 그러나 오른변의 인수가 float형이면 번역프로그람은 1인수구성자를 사용하여 이 float형을 Distance값으로 변경하고 더하기를 한다.

그런데 다음 명령문

```
d3 = 10.0 + d1;
```

은 제대로 동작하지 않는다. 재정의된 +연산자가 성원인 객체는 연산자의 왼변에 변수가 있어야 한다. 여기에 다른 형의 변수나 상수를 배치하면 번역프로그람은 Distance 객체들을 더하는것이 아니라 그 형(이 경우에 float)을 더한다. 이 연산자는 float를 Distance로 변환하는 방법을 모르므로 이 상황에 대처할수 없다. 여기에 실례 11-10의 출력이 있다.

```
d1=2m 50cm
d2=1m 25cm
d3=12m 50cm
```

두번째 더하기는 번역할수 없으므로 여기에 설명문을 붙였다. 이 문제는 Distance형의 새로운 객체를 창조하여 해결할수 있다.

```
d3 = Distance(10,0) + d1;
```

그러나 이것은 적합하지 않다. 연산자의 왼변에 비성원자료형을 가지는 자연스러운 명령문을 쓰려면 어떻게 해야 하겠는가?

바로 friend가 이것을 가능하게 해준다.

(실례 11-11) 재정의된 +연산자의 동료함수

```
#include <iostream>
using namespace std;
class Distance
{
private:
    int meters;
    float centies;
```

```
public:
  Distance() : meters(0), centies(0.0) {}
  Distance(float fltMeters)
  {
    meters = static_cast<int>(fltMeters);
    centies = 100 * (fltMeters - meters);
  }
  Distance(int me, float ce) { meters = me; centies = ce; }
  void ShowDist() { cout << meters << "m " << centies << "cm"; }
  friend Distance operator+(Distance, Distance);
};
Distance operator+(Distance d1, Distance d2)
{
  int m = d1.meters + d2.meters;
  float c = d1.centies + d2.centies;
  if(c >= 100.0) { c -= 100.0; m++; }
  return Distance(m, c);
}
int main()
{
  Distance d1 = 2.5, d2 = 1.25;
  Distance d3;
  cout << "\nd1="; d1.ShowDist();
  cout << "\nd2="; d2.ShowDist();
  d3 = d1 + 10.0;
  cout << "\nd3="; d3.ShowDist();
  d3 =10.0 + d1;
  cout << "\nd3="; d3.ShowDist(); cout << endl;
  return 0;
}
```

실례에서는 재정의된 +연산자를 friend로 만든다.

```
friend Distance operator+(Distance,Distance);
```

성원함수로서 재정의된 +연산자는 1인수를 가지지만 동료함수는 2인수를 가진다. 성원함수에서 +가 연산하는 객체들중 하나는 그것이 성원인 객체이고 다른 하나는 인수이다. 동료함수에서는 둘다 인수이여야 한다.

재정의된 +함수의 본체에 대한 유일한 변경은 실례 11-10에서 객체의 자료를 직접 호출하는데 사용된 변수 meters와 centies가 실례 11-11에서는 d1.meters와 d1.centies로 교체된것이다.

클라스에서 함수선언앞에 friend예약어를 배치하여 함수를 동료로 만든다.

4. 함수표기에서 friend의 사용

friend는 성원함수보다 함수의 호출을 더 명백하게 하는 문법을 제공한다. 실례로 Distance클라스의 객체의 두제곱을 계산하여 float형의 평방메터를 돌려주는 함수를

만들자. 실례 11-12는 이것을 성원함수로 실현하는 방법을 보여준다.

(실례 11-12) Distance의 Square성원함수

```
#include <iostream>
using namespace std;
class Distance
{
private:
  int meters;
  float centies;
public:
  Distance() : meters(0), centies(0.0) {}
  Distance(int me, float ce) { meters = me; centies = ce; }
  void ShowDist() { cout << meters << "m " << centies << "cm"; }
  float Square();
};
float Distance::Square()
{
  float fltMeters = meters + centies / 100;
  float meSqrd = fltMeters * fltMeters;
  return meSqrd;
}
int main()
{
  Distance dist(3, 6.0);
  float sqMe;
  sqMe = dist.Square();
  cout << "\nDistance=";
  dist.ShowDist();
  cout << "\nSquare=" << sqMe << "㎡\n";
  return 0;
}
```

main()은 Distance값을 창조하고 그것을 두제곱하여 결과를 출력한다. 출력은 보통의 거리와 평방메터값이다.

```
Distance=3m 6cm
Square=12.96㎡
```

main()에서 명령문 sqMe = dist.Square();은 dist의 두제곱을 구하여 sqMe에 대입한다. 이것은 옳바로 동작하지만 보통의 수를 사용할 때와 같은 문법으로 Distance객체들과 작업하려고 한다면 함수표기가 더 좋다. 즉

```
sqMe = Square(dist);
```

실례 11-13과 같이 Square()를 Distance의 동료로 만들어 그것을 달성할수 있다.

(실례 11-13) Distance의 Square동료함수

```
#include <iostream>
using namespace std;
class Distance
{
```

```
private:
  int meters;
  float centies;
public:
  Distance() : meters(0), centies(0.0) {}
  Distance(int me, float ce) { meters = me; centies = ce; }
  void ShowDist() { cout << meters << "m " << centies << "cm"; }
  friend float Square(Distance);
};
float Square(Distance d)
{
  float fltMeters = d.meters + d.centies / 100;
  float meSqrd = fltMeters * fltMeters;
  return meSqrd;
}
int main()
{
  Distance dist(3, 6.0);
  float sqMe;
  sqMe = Square(dist);
  cout << "\nDistance=";
  dist.ShowDist();
  cout << "\nSquare=" << sqMe << "㎡\n";
  return 0;
}
```

실례 11-12에서 Square()는 인수가 없는 성원함수이지만 실례 11-13에서는 동료함수로서 하나의 인수를 가진다. 일반적으로 friend함수는 그것이 성원함수일 때보다 인수를 하나 더 가진다. 실례 11-13에서 Square()함수는 실례 11-12와 비슷하지만 meters와 centies와 같은 원천 Distance객체의 자료를 참고한다.

5. 동료클라스

클라스를 동료로 할 때 클라스의 모든 성원함수가 동료로 된다.

(실례 11-14) 동료클라스

```
#include <iostream>
using namespace std;
class Alpha
{
private:
  int data1;
public:
  Alpha() : data1(99) {}
  friend class Beta;
};
class Beta
{
```

```
public:
   void Func1(Alpha a) { cout << "\ndata1=" << a.data1; }
   void Func2(Alpha a) { cout << "\ndata1=" << a.data1; }
};
int main()
{
   Alpha a;
   Beta b;
   b.Func1(a);
   b.Func2(a);
   cout << endl;
   return 0;
}
```

클라스 Alpha에서 클라스 Beta는 동료로 선언된다. 그러면 Beta의 모든 성원함수들은 Alpha의 비공개자료(이 실례에서 data1)를 호출할수 있다.

friend선언에서 Beta는 class예약어를 사용하는 클라스이다.

```
friend class Beta;
```

또한 앞의 실례와 같이 Alpha클라스지정자앞에서 Beta를 클라스로 선언하고 그다음 Alpha안에서 class예약어가 없이

```
friend Beta;
```

와 같은 방법으로 Beta를 참고할수도 있다.

제 3 절. 정적함수

실례 5-18에서 정적자료성원을 소개하였다. 정적자료성원은 객체마다 소유하지 않고 클라스의 모든 객체들이 공유한다. 실례 5-18은 그 클라스의 객체를 몇개나 보유하고있는가를 보여주었다.

자료는 물론 함수도 정적으로 할수 있다. 정적함수를 보여줌으로써 어떤 클라스의 매개 객체에 식별번호를 제공하는 클라스의 모형을 작성한다. 이것은 작성자가 필요한 객체를 찾을수 있게 하고 프로그람의 오유수정에 사용할수 있다. 또한 프로그람에서는 해체자의 조작에 주의를 돌리고있다.

(실례 11-15) 정적함수와 객체의 식별번호

```
#include <iostream>
using namespace std;
class Gamma
{
private:
   static int total;
   int id;
public:
   Gamma() { total++; id = total; }
   ~Gamma()
```

```
    {
        total--;
        cout << id << "번째 객체를 해체하였습니다.\n";
    }
    static void ShowTotal() { cout << "총 개수=" << total << endl; }
    void ShowId() { cout << "식별번호=" << id << endl; }
};
int Gamma::total = 0;
int main()
{
    Gamma g1;
    Gamma::ShowTotal();
    Gamma g2, g3;
    Gamma::ShowTotal();
    g1.ShowId();
    g2.ShowId();
    g3.ShowId();
    cout << "--프로그람 끝--";
    return 0;
}
```

1. 정적함수의 호출

이 프로그람에는 클라스 Gamma에 정적자료성원 total이 있다. 이 자료는 클라스의 객체가 얼마나 있는가를 보관하고 그 값은 구성자에 의해 증가되고 해체자에 의해 감소된다.

클라스밖에서 total을 호출하기 위하여 total값을 출력하는 ShowTotal()이라는 함수를 구성한다.

그러면 이 함수를 어떻게 호출하는가?

자료성원을 static로 선언하면 그 자료값은 클라스전체에 오직 하나만 있으며 그 클라스의 객체를 아무리 많이 창조하여도 값은 같다. 사실상 그러한 객체는 전혀 없을수도 있다. 성원함수호출에 쓰이는 dummyObj를 창조할수 있다. 즉

```
Gamma dummyObj;
dummyObj.ShowTotal();
```

그러나 이 형식은 좋지 않다. 클라스전체와 관련한 작업을 할 때에는 특정한 객체를 참고할 필요가 없으므로 클라스자체의 이름과 범위해결연산자를 사용하는것이 더 좋다.

```
Gamma::ShowTotal();
```

그러나 이 명령문은 일반성원함수인 경우에 제대로 동작하지 않으며 그러한 경우에는 객체와 점성원호출연산자가 요구된다. 클라스이름만 사용하여 ShowTotal()을 호출하려면 실례 11-15처럼 그것을 정적성원함수로 선언해야 한다. 그러면 클라스이름만 사용하여 함수를 호출할수 있다.

아래에 그 출력이 있다.

```
총 개수=1
총 개수=3
식별번호=1
식별번호=2
식별번호=3
--프로그람 끝--
3번째 객체를 해체하였습니다.
2번째 객체를 해체하였습니다.
1번째 객체를 해체하였습니다.
```

어떤 객체 g1을 정의하고 total의 값 1을 출력한다. 그다음 두개의 객체 g2과 g3을 또 정의하고 total의 값 3을 출력한다.

2. 객체에 번호달기

Gamma에서는 개별적객체들이 식별번호를 출력할수 있도록 다른 함수를 정의한다. 식별번호는 객체를 창조할 때 total로 설정되므로 매개 객체는 유일번호를 가진다. ShowTotal()함수는 객체의 식별번호를 출력한다. main()에서는 그것을 세번 호출한다.

```
g1.ShowId();
g2.ShowId();
g3.ShowId();
```

출력할 때 매개 객체는 유일번호를 가지고있다. 즉 g1객체는 1, g2은 2, g3은 3이다.

3. 해체자의 조사

이제는 객체의 수를 알았으므로 해체자에 대한 흥미있는 사실을 조사할수 있다. 실례 11-15는 마지막 명령문에서 "--프로그람 끝--"통보문을 출력하지만 출력이 나타날 때 해체자는 아직 실행되지 않는다. 프로그람에서 창조된 세개의 객체는 프로그람이 완료하기 전에 해체되므로 기억기를 호출할수 없는 상태에 놓이지 않는다. 번역프로그람이 해체자를 호출할 때 이것을 고찰할수 있다.

해체자에 통보문을 출력하는 명령문을 삽입하여 무슨 일이 생기는가를 조사한다. 객체에 번호를 달았으므로 객체들이 파괴되는 순서도 조사할수 있다. 출력이 보여주는 것처럼 마지막으로 창조된 객체 g3이 먼저 파괴된다. 국부객체는 탄창에 보관되므로 LIFO방법에 의하여 마지막으로 창조된것이 먼저 파괴된다.

제 4 절. 대입과 복사초기화

대입연산자와 복사구성자에 대하여 고찰하자.

이미 대입연산자를 많이 사용해왔다.

a1와 a2이 같은 클라스의 객체이라고 하자.

명령문 a2 = a1;는 번역프로그람이 a1의 자료를 한개 성원씩 a2에 복사하게 한다. 이것은 대입연산자의 기정동작이다. 어떤 객체를 다른 객체로 초기화할 때 례를 들면 Alpha a2(a1);는 대입과 비슷한 동작을 한다. 번역프로그람은 새로운 객체 a2을 창조하고 a1로부터 a2에 자료를 하나씩 복사한다. 이것은 복사구성자의 기정동작이다. 이 두가지 기정동작은 번역프로그람에 이미 제공되여있다.

성원별복사를 하려고 한다면 다른 동작을 더 취할 필요가 없다. 그러나 대입이나 초기화를 좀 더 복잡하게 실현하려면 기정함수들을 재정의하여야 한다.

그러면 대입연산자와 복사구성자의 재정의를 따로따로 론의한 다음 String클라스에 기억기를 관리하는 더 효과적인 방법을 제공하는 실례에 그것들을 모두 추가하자.

1. 대입연산자의 재정의

대입연산자를 재정의하는 방법을 보여주는 간단한 실례를 고찰하자.

(실례 11-16) 대입연산자의 재정의

```
#include <iostream>
using namespace std;
class Alpha
{
private:
   int data;
public:
   Alpha() {}
   Alpha(int d) { data = d; }
   void Display() { cout << data; }
   Alpha operator=(Alpha a)
   {
      data = a.data;
      cout << "\n대입연산자를 호출하였습니다.";
      return Alpha(data);
   }
};
int main()
{
   Alpha a1(37);
   Alpha a2;
   a2 = a1;
   cout << "\na2=";
   a2.Display();
   Alpha a3 = a2;
   cout << "\na3=";
   a3.Display();
   cout << endl;
   return 0;
```

```
}
```

Alpha클라스는 아주 간단하며 오직 하나의 자료성원만 포함한다. 구성자들은 자료를 초기화하고 성원함수는 그 값을 출력한다. 실례 11-16에서 새로운것은 대입연산자를 재정의하는 함수 operator=()이다. main()에서는 a1을 정의하고 거기에 값 37을 대입하며 a2을 정의만 하고 값을 주지 않는다. 그다음 대입연산자를 사용하여 a2을 a1의 값으로 설정한다.

```
a2 = a1;
```

여기서 재정의된 operator=()함수가 호출된다. 여기에 실례 11-15의 출력이 있다.

```
대입연산자를 호출하였습니다.
a2=37
a3=37
```

1) 초기화는 대입이 아니다

실례 11-16의 마지막 두 행에서 객체 a3을 a2의 값으로 초기화하고 그것을 표시한다. 여기서 문법을 혼돈해서는 안된다.

```
Alpha a3 = a2; // 대입이 아니라 복사초기화이다.
```

에서 =기호는 대입이 아니라 초기화이고

```
Alpha a3(a2);  // 복사초기화이다.
```

와 같은 효과를 가진다.

이것은 실례 11-16의 출력에서 단일행의

```
대입연산자를 호출하였습니다.
```

에 의해 보여주는것처럼 대입연산자가 오직 1번만 실행되기때문이다.

2) 응답능력얻기

대입연산자를 재정의할 때 기정대입연산자를 실행하였다는 통보를 추가적으로 출력하려고 한다. 대체로 이 조작은 한 객체로부터 다른 객체에로 자료성원들을 복사한다. 실례 11-16의 Alpha클라스는 오직 하나의 자료성원 data를 가지므로 operator=() 함수는 다음 명령문

```
data = a.data;
```

에 의해 값을 복사한다.

또한 이 함수는 실행할 때 "대입연산자를 호출하였습니다."라는 통보를 출력한다.

3) 참고에 의한 넘기기

operator=()의 인수는 참고에 의해 넘어온다. 꼭 그렇게 할 필요는 없으나 일반적으로 아주 좋은 방법이다. 왜서인가?

이미 알고있는것처럼 값에 의해 인수를 넘기면 그것이 넘어가는 함수에서 자체의 사본이 생성된다. operator=()함수에 넘기는 인수도 례외가 아니다. 그러한 객체가 크면 사본은 기억기를 많이 소비한다. 참고에 의해 넘기는 값들은 사본을 생성하지 않으므로 기억기를 절약한다.

또한 객체의 수를 보관해야 하는 경우가 있다. 번역프로그람이 대입연산자를 사용

할 때마다 여분의 객체를 생성하면 필요한것보다 더 많은 객체를 생성한다.

이와 같이 참고에 의한 넘기기는 너무 큰 객체의 창조를 피하게 한다.

4) 값의 돌려주기

이미 알고있는것처럼 함수는 값 또는 참고에 의해 호출측프로그람에 정보를 돌려줄수 있다. 어떤 객체가 값에 의해 귀환할 때 새로운 객체가 창조되여 호출측 프로그람에 돌아온다. 호출측프로그람에서는 그 객체의 값을 새로운 객체에 대입하거나 다른 용도에 사용할수 있다. 그러나 객체가 참고에 의해 복사될 때 새로운 객체는 창조되지 않는다. 함수에서 원시객체에로의 참고는 모두 호출측프로그람에로 돌아온다.

실례 11-16에서 operator=()함수는 림시적인 Alpha객체를 창조하고 명령문

```
return Alpha(data);
```

에서 1인수구성자에 의하여 그것을 초기화하고 값으로 돌려준다.

돌아온 값은 재정의된 대입연산자가 성원인 객체의 사본이고 그것과 같은 객체가 아니다. 값의 돌려주기는 대입연산자를 다중으로 사용할수 있게 한다. 즉

```
a3 = a2 = a1;
```

그러나 값에 의한 귀환은 값에 의해 인수를 넘길 때와 같은 결함을 가지고있다. 즉 여분의 사본을 창조하므로 기억기를 랑비한다.

다음의 선언

```
Alpha& operator=(Alpha& a)  // 이 경우 옳지 않다.
```

와 같이 재정의된 연산자가 참고에 의해 귀환할수 있는가?

함수에서 국부적인 변수에로의 참고귀환을 사용할수 없다. 국부(자동)변수 즉 함수안에서 창조된 변수는 함수로부터 귀환할 때 해체된다. 참고에 의한 귀환은 자료의 주소만 돌려주므로 국부자료는 해체되고 지적자는 무의미한 값으로 된다. 번역프로그람은 이러한 경우에 경고오유를 통보한다.

5) 계승되지 않는 연산자

대입연산자는 계승되지 않는 유일한 연산자이다. 기초클라스에서 대입연산자를 재정의하고 파생클라스에서 이 함수를 사용할수 없다.

2. 복사구성자

이미 알고있는것처럼 객체를 정의하면서 동시에 초기화할수 있다. 즉

```
Alpha a3(a2);          //초기화
Alpha a3 = a2;   //복사초기화
```

이 두가지 정의에서 복사구성자를 호출한다. 복사구성자는 새로운 객체를 창조하고 거기에 인수를 복사하는 구성자이다. 번역프로그람에 의하여 매개 객체에는 기정복사구성자가 자동적으로 제공되고 성원별복사가 진행된다. 이것은 대입연산자가 하는 일과 비슷하다. 차이는 복사구성자가 새 객체를 창조한다는것이다.

대입연산자처럼 복사구성자를 사용자에 의해 재정의할수 있다. 실례 11-17에서 이

것을 보여준다.

(실례 11-17) 복사구성자-X(X&)

```
#include <iostream>
using namespace std;
class Alpha
{
private:
   int data;
public:
   Alpha() {}
   Alpha(int d) { data = d; }
   Alpha(Alpha& a)
   {
      data = a.data;
      cout << "\n복사구성자를 호출하였습니다.";
   }
   void Display() { cout << data; }
   void operator=(Alpha& a)
   {
      data = a.data;
      cout << "\n대입연산자를 호출하였습니다.";
   }
};
int main()
{
   Alpha a1(37);
   Alpha a2;
   a2 = a1;
   cout << "\na2="; a2.Display();
   Alpha a3(a1);
//Alpha a3 = a1;
   cout << "\na3="; a3.Display();
   cout << endl;
   return 0;
}
```

이 프로그람은 대입연산자와 복사구성자를 모두 재정의한다. 재정의된 대입연산자는 실례 11-16과 비슷하다. 복사구성자는 1인수를 가지며 그것은 Alpha형의 객체로 참고에 의해 넘어온다. 그 선언자는

```
Alpha(Alpha &)
```

이 선언자는 X(X&)("X of X ref")형식을 가진다. 여기에 실례 11-17의 출력이 있다.

```
대입연산자를 호출하였습니다.
a2=37
복사구성자를 호출하였습니다.
a3=37
```

명령문 a2 = a1;는 대입연산자를 호출하고 Alpha a3(a1);은 복사구성자를 호출한다. 그와 등가한 명령문 Alpha a3 = a1;도 복사구성자를 호출한다.

이와 같이 객체를 정의할 때 복사구성자를 호출할수 있다. 또한 복사구성자는 인수들이 함수에 값에 의하여 넘어올 때 호출되고 함수로부터 값을 돌려줄 때 호출된다.

1) 함수인수

복사구성자는 객체를 함수에 값에 의해 넘길 때 호출된다. 복사구성자는 그 함수가 조작하는 사본을 창조한다. 따라서 함수 void Func(Alpha);는 XofXref로 선언되고 이 함수가 명령문 Func(a1);에 의해 호출되면 복사구성자가 호출되여 Func()가 사용하는 a1객체의 사본이 창조된다. (물론 복사구성자는 인수가 참고에 의해 넘어가거나 지적자가 넘어오면 호출되지 않는다.)

복사구성자가 창조되지 않는 경우에 함수는 원시변수에 대하여 동작한다.

2) 함수돌림값

또한 복사구성자는 함수로부터 값을 돌려줄 때 림시객체를 창조한다. XofXref에 Alpha Func();와 같은 함수가 있고 명령문 a2 = Func();에 의해 이 함수가 호출되면 복사구성자가 호출되여 Func()에 의해 돌아온 값의 사본이 창조되고 그 값이 대입연산자에 의해 a2에 대입된다.

3) 왜 X(X)구성자가 아닌가

그러면 무엇때문에 복사구성자의 인수를 참고로 사용해야 하는가?

또한 참고대신 값에 의해 넘길수 있는가?

번역프로그람은 Alpha(Alpha a);로 선언하면 "Out of memery"라고 통보한다.

인수를 값에 의해 넘길 때 사본을 구성하기 위하여 복사구성자를 호출한다. 복사구성자를 호출할 때 그 인수를 값에 의해 넘기므로 다시 복사구성자자체를 호출한다. 따라서 복사구성자의 재귀호출이 발생한다. 사실 재귀호출은 번역프로그람이 기억기를 다 써버릴 때까지 진행된다. 따라서 복사구성자에서는 사본을 창조하지 않도록 하기 위하여 인수를 참고에 의해 넘겨야 한다.

4) 해체자와 관련한 주의사항

앞에서 함수에 인수를 값에 의해 넘기는 방법과 값에 의해 돌려주는 방법을 고찰하였다.

이러한 경우는 물론 함수로부터 돌아오는 경우에도 그 함수에 의해 창조된 림시객체들이 파괴될 때 해체자를 호출한다. 이것은 객체의 파괴를 기대하지 않은 경우에 문제로 된다. 그러므로 성원별복사뿐아니라 그 이상의 조작을 요구하는 객체들과 작업할 때에는 될수록 값에 의해서가 아니라 참고에 의하여 넘기고 돌려주어야 한다.

5) 복사구성자와 대입연산자를 둘다 정의한다

대입연산자를 재정의할 때 복사구성자도 재정의하여야 한다. 일반적으로 프로그람 작성자는 자기의 전용복사루틴을 요구하지 않고 기정적으로 주어지는 성원별복사기구

를 사용한다. 성원별복사기구를 명시적으로 사용하지 않는 경우에도 함수에 인수를 값에 의해 넘길 때와 함수로부터 값에 의하여 귀환할 때 암시적으로 사용한다.

사실 어떤 클라스의 구성자가 기억기나 디스크파일과 같은 체계자원의 사용을 포함한다면 대입연산자와 복사구성자를 항상 재정의하고 기대한대로 동작하는가를 확인해야 한다.

6) 복사를 금지하는 방법

대입연산자와 복사구성자를 사용하여 객체들의 복사를 전용화하는 방법을 론의하였다. 그러나 이 조작을 사용하여 객체의 복사를 금지하려고 하는 경우가 있다. 실례로 클라스의 어떤 성원을 구성자에 인수로서 제공되는 성원에 대한 유일한 값으로 창조해야 하는 경우가 있다. 객체를 복사하면 사본에는 같은 값이 주어진다. 복사를 피하려면 대입연산자와 복사구성자를 비공개성원으로 재정의하여야 한다.

```
class Alpha
{
private:
   Alpha& operator=(Alpha&);
   Alpha(Alpha&);
};
```

복사조작을 하면 즉

```
Alpha a1, a2;
a1 = a2;             // 오유: 비공개함수호출
Alpha a3(a1); // 오유: 비공개함수호출
```

라고 실행하면 번역프로그람은 함수를 호출할수 없다고 통보한다. 함수들을 호출하지 못하도록 그것들을 선언만 하고 정의하지 않는것이 좋다.

3. 기억효과문자렬클라스

실제로 실례 11-16과 실례 11-17에서는 재정의된 대입연산자와 복사구성자가 필요없다. 여기서는 하나의 자료항목을 가지는 클라스를 직접 사용하므로 기정대입연산자와 복사구성자를 사용하면 된다. 이 연산자들의 재정의가 중요하게 제기되는 실례를 고찰하자.

1) String클라스를 사용할 때의 결함

앞의 장들에서 String클라스의 여러 판을 보았다. 그러나 이 판들은 그리 복잡하지 않다. 대입연산자를 재정의하여 어떤 String객체의 값을 다른 객체에 대입할수 있다. 즉

```
s2 = s1;
```

대입연산자를 재정의하면 String클라스의 기본자료항목인 실제문자렬(char형배렬)의 관리문제가 제기된다.

하나의 가능성은 매개의 String객체가 문자렬을 보관하는 장소를 가리키는것이다. 어떤 String객체를 다른 객체에 대입하면 단순히 원천객체로부터 목적객체에로 문자렬

을 복사한다. 기억기절약과 관련되여있으면 그와 관련한 문제는 현재 같은 문자렬이 기억기의 두 곳이상에 존재하는것이다. 이것은 매우 비효과적이며 특히 문자렬이 길수록 더해진다. 그림 11-4에 그것을 보여준다.

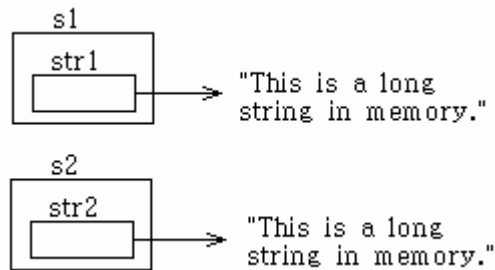



그림 11-4. 문자렬의 복사

여러개의 String객체에 자기의 char*문자렬을 보관할 대신 문자렬에로의 지적자를 포함한다. 한개 String객체를 다른 객체에 대입하면 한 객체로부터 다른 객체에 지적자만 복사하므로 두 지적자는 같은 문자렬을 가리키게 된다. 이것은 효과적이다. 그것은 문자렬의 오직 하나의 부분만 기억기에 보관되기때문이다. 그림 11-5는 이것을 보여준다.

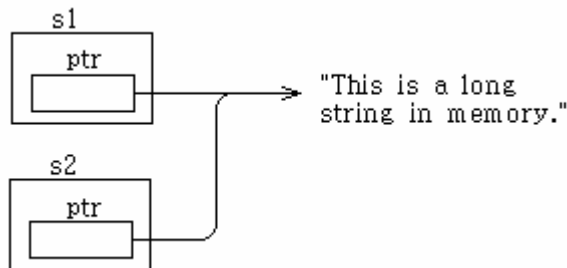



그림 11-5. 문자렬의 복사

그러나 이 방법을 사용하면 String객체를 해체할 때 주의해야 한다. String의 해체자가 delete를 사용하여 문자렬이 차지한 기억기를 해방하면 그 문자렬을 가리키는 지적자를 가지는 객체가 여러개 있을 때 다른 객체들은 문자렬을 더는 가리키지 않는 지적자를 보유하게 된다. 즉 그것들은 종속적인 지적자로 된다.

String객체의 문자렬들이 지적자를 사용하려면 특정한 문자렬을 지적하는 String객체의 수를 보관하고있다가 문자렬을 가리키는 마지막 String객체가 해체될 때에만 문자렬을 삭제하게 하는 방법이 요구된다. 다음의 실례 11-18에 이것을 보여준다.

2) 문자렬계수기클라스

같은 문자렬을 가리키는 여러개의 String객체가 있다고 하자.

그리고 그 문자렬을 가리키는 객체의 수를 보관하려고 한다. 어디서 그것을 계수하겠는가?

매개의 String객체가 특정문자렬을 가리키는 String객체의 수를 소유하는것은 시끄

러운 일이다. 따라서 String안에서 계수용성원변수를 사용하지 않고 정적변수를 사용할수 있다. 즉 정적배렬을 창조하고 그것을 문자렬주소와 개수의 보관에 사용할수 있다. 그러나 이것은 상당한 품을 요구한다. 그러므로 수량을 보관하는 클라스를 새로 창조하는것이 더 효과있다. StrCount라는 클라스의 매개 객체는 계수값과 그 문자렬자체에로의 지적자를 포함한다. 매개 String객체는 적당한 StrCount객체에로의 지적자를 포함한다. 그림 11-6은 이것을 보여준다.

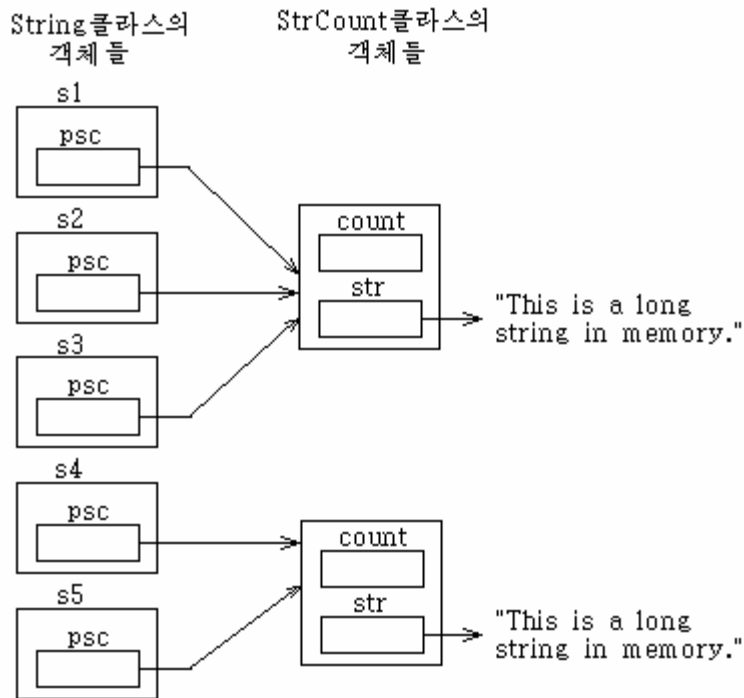



그림 11-6. String과 StrCount클라스

String객체가 StrCount클라스의 객체들을 호출할수 있도록 담보하기 위하여 String을 StrCount의 동료로 만든다. 또한 StrCount클라스가 String클라스에 의해서만 사용된다는것을 담보하여야 한다. 임의의 함수에 대한 호출을 방지하기 위하여 StrCount의 모든 성원함수를 비공개로 한다. String이 동료이므로 StrCount의 임의의 부분을 String만이 호출할수 있다.

여기에 실례 11-18의 프로그람이 있다.

(실례 11-18) 기억기절약문자렬클라스-대입연산자와 복사구성자의 재정의

```
#include <iostream>
#include <cstring>
using namespace std;
class StrCount
{
private:
   int count;
```

```
    char* str;
    friend class String;
    StrCount(char* s)
    {
        int length = strlen(s);
        str = new char[length + 1];
        strcpy(str, s);
        count = 1;
    }
    ~StrCount()
    {
        delete[] str;
    }
};
class String
{
private:
    StrCount* psc;
public:
    String() { psc = new StrCount("NULL"); }
    String(char* s) { psc = new StrCount(s); }
    String(String& s)
    {
        psc = s.psc;
        (psc->count)++;
    }
    ~String()
    {
        if(psc->count == 1)
            delete psc;
        else
            (psc->count)--;
    }
    void Display()
    {
        cout << psc->str;
        cout << " (주소=" << psc << ")";
    }
    void operator=(String& s)
    {
        if(psc->count == 1)
            delete psc;
        else
            (psc->count)--;
        psc = s.psc;
        (psc->count)++;
    }
};
```

```
int main()
{
  String s3 = "이것은 문자렬이다.";
  cout << "\ns3="; s3.Display();
  String s1;
  s1 = s3;
  cout << "\ns1="; s1.Display();
  String s2(s3);
  cout << "\ns2="; s2.Display();
  cout << endl;
  return 0;
}
```

main()에서 String객체 s3을 정의하고 여기에 "이것은 문자렬이다."라는 문장을 포함한다. 또 다른 String객체 s1를 정의하고 그것을 s3과 같게 설정한다. 그다음 s2을 정의하고 s3으로 초기화한다. s1를 s3과 같게 설정하면 재정의된 대입연산자가 호출되고 s2을 s3으로 초기화하면 재정의된 복사구성자가 호출된다. 세개의 문자렬을 모두 출력하고 매개 객체의 psc지적자가 가리키는 StrCount객체의 주소를 출력하면 이 객체들이 모두 같다는것을 알수 있다. 실례 11-18의 출력은 다음과 같다.

```
s3=이것은 문자렬이다.(주소=0x88510e00)
s1=이것은 문자렬이다.(주소=0x88510e00)
s2=이것은 문자렬이다.(주소=0x88510e00)
```

이렇게 문자렬클라스를 String과 StrCount클라스로 분할하였다.

3) StrCount클라스

StrCount클라스는 실제문자렬에로의 지적자와 그 문자렬을 가리키는 String클라스의 객체수를 포함한다. StrCount의 유일한 구성자는 문자렬에로의 지적자를 인수로 가지고 문자렬용의 새로운 기억령역을 창조한다. 구성자는 문자렬을 이 령역에 복사하고 그것이 창조될 때 한개의 String객체를 가리키므로 count를 1로 설정한다. StrCount의 해체자는 문자렬이 사용하는 기억기를 해방한다.

4) String클라스

String클라스는 세개의 구성자를 사용한다. 새 문자렬이 (0인수, 1인수구성자에 의해) 창조되면 새로운 StrCount객체가 창조되고 문자렬을 보관하며 psc지적자는 그 객체를 가리키도록 설정된다. 현존 String객체가 (복사구성자와 재정의된 대입연산자에 의해) 복사되면 지적자 psc는 이전의 StrCount객체를 가리키도록 설정되고 그 객체의 계수값을 증가시킨다.

해체자는 물론 재정의된 대입연산자는 count가 1이면 psc가 가리키는 이전의 StrCount객체를 삭제해야 한다. (하나의 StrCount객체를 삭제하므로 delete에 괄호를 쓰지 않는다.)

그러면 무엇때문에 대입연산자에서 삭제를 고려하여야 하는가?

=기호의 왼변에 있는 String객체(s1)에 대입하기 전에 어떤 StrCount객체

(oldStrCut)을 가리키고있다. 대입후에 s1은 =기호의 오른변에 있는 객체를 가리킨다. 현재 oldStrCut를 가리키는 String객체가 없다면 그것을 삭제해야 한다. 또한 그것을 가리키는 다른 객체가 있다면 그 계수량을 감소시켜야 한다. 그림 11-7은 재정의된 대입연산자의 동작을, 그림 11-8은 복사구성자의 동작을 보여준다.

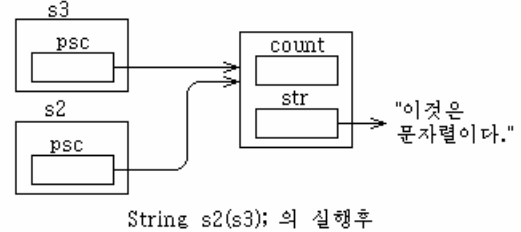
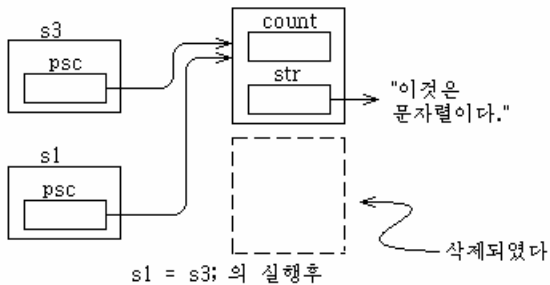
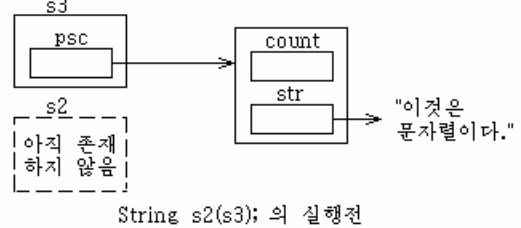
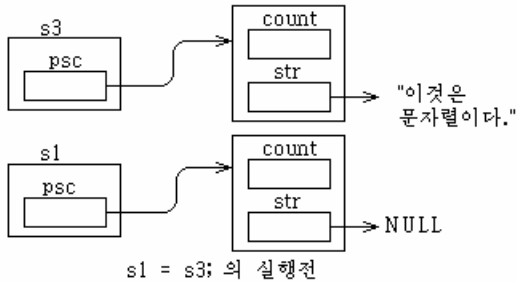



그림 11-7. 실례 11-18의 대입연산자 그림 11-8. 실례 11-18에서 복사구성자

제 5 절. this지적자

매개 객체의 성원함수는 this라는 신기한 지적자의 호출을 가진다. this는 객체자체를 가리킨다. 그러므로 임의의 성원함수는 그것이 성원으로 되는 객체의 주소를 찾을 수 있다. 여기에 간단한 실례가 있다.

(실례 11-19) this지적자

```
#include <iostream>
using namespace std;
class Where
{
private:
   char chArray[10];
public:
   void Reveal() { cout << "\n이 객체의 주소=" << this; }
};
int main()
{
   Where w1, w2, w3;
   w1.Reveal();
```

```
    w2.Reveal();
    w3.Reveal();
    cout << endl;
    return 0;
}
```

실례에서 main()프로그람은 Where형의 세개 객체를 창조한다. 그다음 매개 객체는 this지적자의 값을 출력하는 Reveal()성원함수에 의하여 그 주소를 출력한다. 출력은 다음과 같다.

```
이 객체의 주소는 0x8f4effec
이 객체의 주소는 0x8f4effe2
이 객체의 주소는 0x8f4effd8
```

매개 객체의 자료는 10byte의 배렬로 이루어지므로 객체들을 기억기에 10byte간격으로 보관한다. 일부 번역프로그람은 객체에 여유바이트를 배치하는 경우도 있으므로 10byte를 넘을수 있다.

1. this에 의한 성원자료의 호출

성원함수를 호출하면 호출된 객체의 주소로 설정되는 this의 값이 존재하게 된다. this지적자를 객체에로의 지적자처럼 다룰수 있고 그것이 가리키는 객체의 자료를 호출하는데 사용할수 있다. 실례 11-20에 이것을 보여준다.

(실례 11-20) this지적자에 의한 자료의 참고

```
#include <iostream>
using namespace std;
class What
{
private:
    int alpha;
public:
    void Tester()
    {
        this->alpha = 11;
     // alpha = 11;
        cout << this->alpha;
     // cout << alpha;
    }
};
int main()
{
    What w;
    w.Tester();
    cout << endl;
    return 0;
}
```

프로그람은 값 11을 출력한다. Tester()성원함수는 변수 alpha를 this-> alpha와

같이 호출한다. 이것은 alpha에로의 직접참고와 같다. 이러한 문법은 옳으며 this가 실제로 객체를 지적한다는것을 보여준다.

2. 돌림값에 this의 사용

this의 실천적인 사용은 성원함수와 재정의된 연산자로부터 값을 돌려주는 경우이다. 실례 11-16에서는 참고에 의해 객체를 돌려줄수 없다. 그것은 함수로부터 귀환할 때 객체가 국부객체이므로 해체되기때문이다. 참고에 의해 객체를 돌려주려면 더 영속적인 객체가 요구된다. 어떤 함수를 성원으로 가지는 객체는 그 개별적인 성원함수들보다 영속적이다. 객체의 성원함수는 그것이 호출될 때마다 창조되고 해체되지만 객체자체는 그것이 어떤 다른것에 의해 해체될 때까지(즉 그것이 삭제될 때까지) 존재한다.

이처럼 어떤 함수가 성원으로 되여있는 객체를 참고에 의해 돌려주는것은 성원함수안에서 창조된 림시객체를 돌려주는것보다 더 좋다. 바로 this지적자가 이것을 가능하게 한다.

실례 11-21에서 operator=()함수는 그것을 호출한 객체를 참고에 의해 돌려준다.

(실례 11-21) this지적자의 내용을 돌려주기

```
#include <iostream>
using namespace std;
class Alpha
{
private:
   int data;
public:
   Alpha() {}
   Alpha(int d) { data = d; }
   void Display() { cout << data; }
   Alpha& operator=(Alpha& a)
   {
      data = a.data;
      cout << "\n대입연산자를 호출하였습니다.";
      return *this;
   }
};
int main()
{
   Alpha a1(37);
   Alpha a2, a3;
   a3 = a2 = a1;
   cout << "\na2=";
   a2.Display();
   cout << "\na3=";
   a3.Display();
   cout << endl;
   return 0;
```

```
}
```

프로그람에서는 값을 돌려주는 Alpha operator=(Alpha a)대신에 참고를 돌려주는 선언 Alpha& operator=(Alpha a);를 사용한다.

이 함수의 마지막 명령문은 return *this;이다.

this는 함수가 성원인 객체에로의 지적자이므로 *this는 그 객체이며 이 명령문은 그것을 참고에 의해 돌려준다. 여기에 실례 11-21의 출력이 있다.

```
대입연산자를 호출하였습니다.
a2=37
a3=37
```

=기호가 a3 = a2 = a1;과 같이 식에 나타날 때마다 재정의된 operator=()함수가 호출되고 통보문이 출력된다. 세개의 객체는 모두 같은 값을 가진다.

일반적으로 재정의된 대입연산자로부터 *this에 의하여 참고를 돌려줌으로써 여유객체의 창조를 피할수 있다.

3. 수정한 기억기절약문자렬클라스

this지적자를 사용하여 실례 11-18의 operator=()함수를 참고에 의해 값을 돌려주도록 변경하여 String객체들의 다중대입연산이 가능하게 할수 있다. 즉

```
s1 = s2 = s3;
```

이와 함께 값에 의하여 귀환할 때 막대한 량의 객체가 창조되는것을 방지하게 한다. 여기에 실례 11-22의 프로그람이 있다.

(실례 11-22) 기억기절약문자렬클라스 - 대입연산자에서 this지적자의 사용

```
#include <iostream>
#include <cstring>
using namespace std;
class StrCount
{
private:
   int count;
   char* str;
   friend class String;
   StrCount(char* s)
   {
      int length = strlen(s);
      str = new char[length + 1];
      strcpy(str, s);
      count = 1;
   }
   ~StrCount()
   { delete[] str; }
};
class String
{
```

```
private:
  StrCount* psc;
public:
  String() { psc = new StrCount("NULL"); }
  String(char* s) { psc = new StrCount(s); }
  String(String& s)
  {
     cout << "\n복사구성자";
     psc = s.psc;
     (psc->count)++;
  }
  ~String()
  {
     if(psc->count == 1)
         delete psc;
     else
         (psc->count)--;
  }
  void Display()
  {
     cout << psc->str;
     cout << " (주소=" << psc << ")";
  }
  String& operator=(String& s)
  {
     cout << "\n대입";
     if(psc->count == 1)
         delete psc;
     else
         (psc->count)--;
     psc = s.psc;
     (psc->count)++;
     return *this;
  }
};
int main()
{
  String s3 = "이것은 문자렬이다.";
  cout << "\ns3="; s3.Display();
  String s1, s2;
  s1 = s2 = s3;
  cout << "\ns1="; s1.Display();
  cout << "\ns2="; s2.Display();
  cout << endl;
  return 0;
}
```

대입연산자를 위한 선언자는 다음과 같다.

```
String & operator=(String& s)
```

그리고 실례 11-21과 같이 이 함수는 this에로의 지적자를 돌려준다.

실행결과는 다음과 같다.

```
이것은 문자렬이다.
대입
대입
s1=이것은 문자렬이다.
s1=이것은 문자렬이다.
```

출력은 대입명령문에 의해 세개의 String객체가 같은 StrCount객체를 가리킨다는 것을 보여준다.

this지적자는 정적성원함수들에서 비효과적이다. 그것은 정적성원함수가 특정한 객체와 련관되여있기때문이다.

- 자체대입에서 주의할 점

재정의된 대입연산자가 있으면 어떤 객체를 그 자체로 설정하는데 사용할수 있다.

```
alpha = alpha;
```

재정의된 대입연산자는 이와 같은 자체대입을 조종하도록 준비되여야 한다. 한편 좋지 않은 일이 생길수 있다. 실례 11-22의 main()부분에서 StrCount객체를 사용하는 다른 String객체들이 없는 경우에 String객체를 그 자체로 설정하면 프로그람은 강제로 중단된다. 문제는 대입연산자의 코드에서 StrCount객체를 호출한 객체가 StrCount객체를 사용하는 유일한 객체이라면 그 StrCount객체를 삭제하는데 있다. 즉 자체대입에 의하여 그 객체는 삭제된다.

이 문제를 해결하려면 재정의된 대입연산자의 선두에서 자체대입을 검사해야 한다.

대부분의 경우에 연산자를 호출한 객체의 주소와 인수의 주소를 비교하고 주소가 같으면 객체들은 등가하므로 즉시 돌려주어야 한다. (이미 같으므로 서로 대입할 필요가 없다.) 실례 11-20에서 operator=()의 선두에 다음 행을 삽입할수 있다.

```
if (this == &s)
   return *this;
```

제 6 절. 동적형정보

객체의 클라스에 대한 정보는 실행시에 객체의 클라스가 변할 때에도 얻을수 있다. 우리는 주로 두개의 기구 즉 dyname_cast연산자와 typeid연산자를 고찰한다. 이것들은 큰 능력을 가지며 그것이 필요한 경우가 있다.

dyname_cast와 typeid의 능력은 보통 기초클라스로부터 각종 클라스들을 파생시킬 때 발휘된다. 동적강제형변환이 가능하려면 기초클라스는 다형적이여야 한다. 즉 적어도 하나의 가상함수를 가져야 한다.

dyname_cast와 typeid가 모두 동작하려면 번역프로그람은 실행시 형정보를 가능하게 해야 한다. Boland C++ Builder는 기정으로 이 기능이 가능하게 되여있다.

Microsoft Visual C++에서는 그것을 설정해야 한다. 또한 머리부파일 typeinto를 포함해야 한다.

1. dyname_cast에 의한 클라스의 형검사

어떤 프로그람이 작성자의 프로그람에 어떤 객체를 보낸다고 가정하자.(이것은 조작체계와 같이 역호출함수를 사용하면 가능하다.)

그것이 일정한 형의 객체이라고 가정하자. 그리고 그것이 정확한가 검사하려고 한다.

그러면 어떤 객체의 형을 어떻게 알겠는가?

바로 dyname_cast연산자는 한가지 방법을 준다. 여기서는 검사하려는 객체들이 모두 공통의 선조로부터 계승되는것으로 가정한다. 실례 11-23에서 이것을 보여준다.

(실례 11-23) 객체의 형검사에 쓰이는 동적객체만들기

```
// RTTI를 가능으로 설정해야 한다.
#include <iostream>
#include <typeinfo>
using namespace std;
class Base
{
public:
   virtual void VirtFunc() { }
};
class Derv1 : public Base {};
class Derv2 : public Base {};
bool IsDerv1(Base* pUnknown)
{
   Derv1* pDerv1;
   if(pDerv1 = dynamic_cast<Derv1*>(pUnknown))
      return true;
   else
      return false;
}
int main()
{
   Derv1* d1 = new Derv1;
   Derv2* d2 = new Derv2;
   if(IsDerv1(d1))
      cout << "d1은 Derv1클라스의 성원입니다.\n";
   else
      cout << "d1은 Derv1클라스의 성원이 아닙니다.\n";
   if(IsDerv1(d2))
      cout << "d2은 Derv1클라스의 성원입니다.\n";
   else
      cout << "d2은 Derv1클라스의 성원이 아닙니다.\n";
   return 0;
```

```
}
```

여기에 하나의 기초클라스 Base와 두개의 파생클라스 Derv1과 Derv2가 있다. 또한 함수 IsDerv1()이 있다. IsDerv1()함수는 인수로 받아들인 지적자가 Derv1클라스의 객체를 가리킬 때 true를 돌려준다. 인수는 클라스 Base형이므로 넘어온 객체는 Derv1 또는 Derv2일수 있다. dyname_cast연산자는 알려지지 않은 지적자 pUnknown을 Derv1형으로 변환하려고 시도한다. 그 결과가 0이 아니면 pUnkown은 Derv1객체를 지적하고 결과가 0이면 다른것을 지적한다.

2. dyname_cast에 의한 지적자형의 변경

동적강제형변환연산자는 계승나무에서 우로 혹은 아래로 강제형변환하게 한다. 그러나 이러한 강제형변환은 제한된 방법으로만 가능하다. 실례 11-24는 강제형변환실례를 보여준다.

(실례 11-24) 동적강제형변환

```
// RTTI를 가능으로 설정해야 한다.
#include <iostream>
#include <typeinfo>
using namespace std;
class Base
{
protected:
   int ba;
public:
   Base() : ba(0) {}
   Base(int b) : ba(b) {}
   virtual void VirtFunc() { }
   void Show() { cout << "Base: ba=" << ba << endl; }
};
class Derv : public Base
{
private:
   int da;
public:
   Derv(int b, int d) : da(d) { ba = b; }
   void Show() { cout << "Derv: ba=" << ba << ", da=" << da << endl; }
};
int main()
{
   Base* pBase = new Base(10);
   Derv* pDerv = new Derv(21, 22);
   // 파생으로부터 기초에로의 올리강제형변환
   pBase = dynamic_cast<Base*>(pDerv);
   pBase->Show();
   pBase = new Derv(31, 32);
   // 기초로부터 파생에로의 내리강제형변환
```

```
    pDerv = dynamic_cast<Derv*>(pBase);
    pDerv->Show();
    return 0;
  }
```

실례에는 기초클라스와 파생클라스가 각각 하나씩 있다. 매개 클라스에는 강제형변환의 효과를 보여주는 자료항목이 하나씩 있다.

올리강제형변환(up cast)에서는 파생클라스객체를 기초클라스객체로 변환하려고 시도한다. 이때 얻어지는것은 파생클라스객체의 기초부분이다. 실례에서는 클라스 Derv의 객체를 만든다. 이 객체의 기초클라스부분은 값 21을 가지는 성원자료 ba이고 파생부분은 값 22을 가지는 자료성원 da를 보유한다. 강제형변환후 pBase는 이 Derv클라스객체의 기초클라스부분을 지적하므로 그 자체를 표시하기 위해 호출했을 때 Base:ba=21을 출력한다. 올리강제형변환은 객체의 기초부분을 요구할 때 사용하는것이 좋다. 내리강제형변환(down cast)에서는 기초클라스지적자가 가리키는 파생클라스객체를 파생클라스지적자에 넣을수 있다.

3. typeid연산자

때때로 객체의 클라스형을 확인하는것보다 그 객체에 대한 구체적인 정보를 요구하는 경우가 있다. typeid연산자에 의하여 알려지지 않은 객체의 클라스이름과 같은 정보를 얻을수 있다. 실례 11-25에서 그것을 보여준다.

(실례 11-25) typeid함수

```
// RTTI를 가능으로 설정해야 한다.
#include <iostream>
#include <typeinfo>
using namespace std;
class Base
{
public:
  virtual void VirtFunc() { }
};
class Derv1 : public Base {};
class Derv2 : public Base {};
void DisplayName(Base* pB)
{
  cout << typeid(*pB).name();
  cout << "의 객체에로의 지적자\n";
}
int main()
{
  Base* pBase = new Derv1;
  DisplayName(pBase);
  pBase = new Derv2;
  DisplayName(pBase);
```

```
        return 0;
    }
```

실례에서 DisplayName()함수는 거기에 넘어온 객체의 클라스이름을 표시하기 위하여 typeid연산자자체의 type_info클라스의 name()성원을 사용한다. main()에서는 typeid()함수에 클라스 Derv1과 클라스 Derv2의 객체들을 하나씩 넘긴다. 이때 프로그람의 출력은 다음과 같다.

```
class Derv1의 객체에로의 지적자
class Derv2의 객체에로의 지적자
```

그 이름외에도 클라스에 대한 다른 정보를 typeid에 의해 얻을수 있다. 실례로 재정의된 ==연산자를 사용하여 클라스의 동등성을 검사할수 있다. 실례 12-11에서 이것을 보여준다.

이 절의 실례에서는 지적자를 사용하지만 dynamic_cast와 typeid는 참고인 경우에도 잘 동작한다.

요 약

가상함수는 프로그람을 실행할 때 어떤 함수를 호출하는가를 결정하는 방도를 준다. 보통 이러한 결정은 번역시에 이루어진다. 가상함수는 각종 객체에 대하여 같은 종류의 동작을 처리하는데서 더 큰 융통성을 준다. 특히 여러가지 파생형들에로의 지적자(또는 참고)를 실제로 보관하는 지적자형의 배렬로부터 함수들의 호출을 허용한다. 이것은 다형성의 하나의 실례이다. 일반적으로 함수를 기초클라스에서 가상으로 선언하고 같은 이름을 가지는 다른 함수를 파생클라스들에서 선언한다.

한개이상의 순수가상함수를 클라스에서 선언하면 그 클라스는 추상클라스로 된다. 이것은 어떤 객체도 그의 실례로 될수 없다는것을 의미한다.

동료함수는 클라스의 비공개자료를 호출할수 있다. 클라스의 성원함수가 아니라도 동료함수는 어떤 함수가 두개이상의 비련관된 클라스들을 호출해야 할 때와 재정의된 연산자가 그것이 성원으로 되지 않는 다른 클라스의 값을 그 왼변에서 사용해야 할 때 효과가 있다.

일반적으로 정적함수는 클라스의 객체보다도 클라스에 대하여 동작하는 함수이다. 특히 정적함수는 정적변수에 대해 동작할수 있다. 정적함수는 클라스이름과 범위해결연산자를 사용하여 호출한다.

대입연산자를 재정의할수 있다. 대입연산자는 어떤 객체의 내용을 다른 객체에 완전히 복사할 때 요구된다. 복사구성자는 초기화할 때 사본을 창조하고 함수에 인수를 넘기거나 값에 의해 함수로부터 귀환할 때 호출된다. 복사구성자는 어떤 객체를 완전히 복사할 때 요구된다.

this지적자는 함수가 성원으로 되는 객체를 가리키도록 성원함수에 미리 정의된다.

this지적자는 그 함수가 성원인 객체를 돌려줄 때 사용된다.

dynamic_cast연산자는 여러가지 역할을 수행한다. 이 연산자는 지적자가 어떤 객체의 형을 지적하는가를 결정하는데 쓰이며 일정한 경우에 지적자의 형을 변경할수 있다. typeid연산자는 객체의 클라스에 대한 일정한 정보(실례로 클라스의 이름)를 얻을수 있다.

문 제

1. 가상함수는

① 파생클라스들에로의 지적자들을 보관할수 있는 지적자기초클라스형의 배렬을 참고하게 한다.

② 절대로 호출할수 없는 함수들을 창조하게 한다.

③ 다른 클라스들의 객체들을 묶어서 같은 함수코드에 의해 객체들을 호출할수 있게 한다.

④ 같은 함수호출에 의하여 각이한 클라스의 객체들의 성원함수들을 호출하게 한다.

어느것이 옳은가?

2. 기초클라스에로의 지적자는 파생클라스의 객체들을 가리킬수 있는가?

3. 기초클라스객체에로의 지적자 p가 있고 그것이 파생클라스의 객체의 주소를 포함할 때 두 클라스들이 비가상성원함수 Ding()을 포함한다면 명령문 p->Ding();은 어느 클라스의 Ding()을 호출하는가?

4. void형을 돌려주고 int형인수를 하나 가지는 Dang()이라는 가상함수용선언자를 쓰시오.

5. 프로그람이 실행을 시작한 후에 특정함수호출명령문에 의하여 어느 함수가 실행되는가를 결정하는것을 무엇이라고 하는가?

6. 기초클라스의 객체에로의 지적자 p가 있고 그것이 파생클라스의 객체의 주소를 포함하며 두 클라스들이 가성성원함수 Ding()을 포함한다면 명령문 p->Ding();은 어느 클라스의 Ding()을 실행하는가?

7. 값을 돌려주지 않고 인수를 가지지 않는 순수가상함수 Aragorn의 선언을 쓰시오.

8. 순수가상함수는

① 그 클라스를 추상화하게 하는 가상함수이다.

② 아무것도 돌려주지 않는 가상함수이다.

③ 기초클라스로부터 사용되는 가상함수이다.

④ 인수를 가지지 않는 가상함수이다.

어느것이 옳은가?

9. 클라스 Dong의 객체들에로의 10개 지적자들의 배렬 pArr의 정의를 쓰시오.

10. 추상클라스는

① 어떤 클라스도 그로부터 파생되지 않아야 할 때

② 한 파생클라스로부터 다른 파생클라스에로의 여러 경로가 있을 때

③ 그로부터 어떤 객체의 실례도 만들지 않을 때

④ 클라스의 선언을 지연시키려고 할 때 사용할수 있다.

어느것이 옳은가?

11. 동료함수는 클라스성원이 아니지만 클라스의 비공개자료를 호출할수 있는가?

12. 동료함수는

① 클라스들사이에 인수들을 중개하는데

② 원천코드가 유효하지 않는 클라스들을 호출하는데

③ 련관이 없는 클라스에로 호출하려고 할 때

④ 재정의된 연산자의 만능성을 증가시키려고 할 때 사용할수 있다.

어느것이 옳은가?

13. void형을 돌려주는 클라스 George의 한개 인수를 가지는 동료함수 Harry()의 선언을 쓰시오.

14. 예약어 friend는

① 한 클라스가 다른 클라스를 호출하게 할 때 포함한다.

② 한 클라스가 다른 클라스에로의 호출을 요구할 때 포함한다.

③ 클라스의 비공개부분에 있다.

④ 클라스의 공개부분에 있다.

어느것이 옳은가?

15. 클라스 Harry의 매개 성원들을 동료함수로 만드는 선언을 쓰시오.

16. 정적함수는

① 객체가 파괴될 때 호출되여야 한다.

② 클라스의 개별적인 객체와 밀접히 련결된다.

③ 클라스이름과 함수이름을 사용하여 호출할수 있다.

④ 무효객체를 창조해야 할 때 사용된다.

어느것이 옳은가?

17. 기정대입연산자 =를 객체들에 적용할 때 무엇을 하는가 설명하시오.

18. 클라스 Zeta에서 재정의된 대입연산자의 선언을 쓰시오.

19. 대입연산자는

① 동일한 객체들의 번호의 자리길을 유지하는것을 방조하기 위하여

② 매개 객체에 따로따로의 식별번호를 붙이기 위하여

③ 모든 성원자료들이 정확히 복사된다는것을 담보하기 위하여

④ 언제 대입이 진행되는가를 알리기 위하여 재정의하여야 한다.

어느것이 옳은가?

20. 사용자는 항상 복사구성자의 조작을 정의해야 한다. 옳은가?

21. 대입연산자와 복사구성자의 조작은

① 복사구성자가 새로운 객체를 창조하는것을 제외하고 서로 비슷하다.

② 대입연산자가 성원자료를 복사하는것을 제외하고 서로 비슷하다.

③ 그것들이 둘다 새로운 객체를 창조하는것을 제외하고 서로 다르다.

④ 대입연산자가 성원자료를 복사하는것을 제외하고 서로 다르다.

어느것이 옳은가?

22. Bartha클라스용 복사구성자의 선언을 쓰시오.

23. 복사구성자는 객체의 자료부분만 복사하기 위하여 정의할수 있다. 옳은가?

24. 변수의 수명은

① 성원함수가 함수의 수명과 일치하면 자동으로 정의한다.

② 클라스의 수명과 일치하면 외부로 정의한다.

③ 객체의 비정적성원자료는 객체의 수명과 일치할 때 정의한다.

④ 성원함수에서 함수의 수명과 일치하면 정적으로 정의한다.

어느것이 옳은가?

25. 값에 의해 돌려줄 때와 마찬가지로 성원함수안에서 정의된 자동변수의 값을 돌려보내는것과 관련하여 어떤 문제가 제기되는가?

26. 다음의 두개 명령문사이의 조작상 차이를 말하시오.

```
Person p1(p0);
Person p1 = p0;
```

27. 복사구성자는

① 함수가 값에 의하여 귀환할 때

② 인수가 값에 의하여 넘어올 때

③ 함수가 참고에 의하여 귀환할 때

④ 인수가 참고에 의하여 귀환할 때 호출된다.

어느것이 옳은가?

28. this지적자는 무엇을 가리키는가?

29. da가 클라스의 성원변수라면 명령문 this.da = 37;은 da에 37을 대입하는가?

30. 성원함수가 림시객체를 창조하지 않고 그것이 성원으로 되는 객체를 돌려주는데 사용할수 있는 명령문을 쓰시오.

련습문제

1. 9장의 련습 1과 같은 음성카세트판을 만든다고 하자. 출판물의 제목(string)과 가격(float형)을 보관하는 Publication이라는 클라스를 창조하고 이 클라스로부터 두개의 클라스를 파생하시오. Book에는 페지수(int형)를 추가하고 Tape에는 연주시간(분, float형)을 추가한다. 매개 클라스는 GetData()함수를 사용하여 건반으로부터 객체의 자료를 얻으며 PutData()함수는 자료를 표시한다. main()함수에서 Publication에로의 지적자배렬을 창조하시오. 실례 11-4와 비슷하게 순환에서 사용자에게 특정한 도서와 테프에 대한 자료를 입력하게 하고 new에 의하여 그 자료에 기초하여 Book 또는 Tape형객체를 창조하시오. 사용자가 모든 도서와 테프의 자료를 입력한 다음 입력한 도서와 테프의 결과자료를 표시하시오. 이때 for순환과

```
pubArr[j]->PutData();
```

와 같은 단일명령문을 사용하여 배렬의 매개 객체로부터 자료를 표시하시오.

2. 실례 11-11과 실례 11-13과 같이 Distance클라스에서 두개의 거리를 곱하는 연산자를 창조하시오. 그리고 그것을 friend함수로 하여 다음 식에서 사용하도록 하시오.

```
wdist1 = 7.5 * dist2;
```

류동소수점수값을 Distance값으로 변환하는데 1인수구성자를 사용하시오. 여러가지 방법으로 이 연산자를 시험하는 main()함수를 쓰시오.

3. 배렬처럼 동작하는 클라스를 만들수 있다. 실례 11-26은 자체의 배렬클라스를 창조하는 한가지 방법을 보여준다.

(실례 11-26) 배렬클라스의 창조

```
#include <iostream>
using namespace std;
class Array
{
private:
   int* ptr;
   int size;
public:
   Array(int s)
   {
      size = s;
      ptr = new int[s];
   }
   ~Array()
   {
      delete[] ptr;
   }
   int& operator[](int j)
```

```
    { return *(ptr + j); }
};
int main()
{
   const int ASIZE = 10;
   Array arr(ASIZE);
   for(int j=0; j<ASIZE; j++)
      Arr[j] = j * j;
   for(j=0; j<ASIZE; j++)
      cout << arr[j] << ' ';
   cout << endl;
   return 0;
}
```

이때 프로그람의 출력은 다음과 같다.

```
0   1   4   9  16  25  36  49  64  81
```

4. 실례 11-26에서 재정의된 대입연산자와 재정의된 복사구성자를 Array클라스에 추가하시오. 그다음 main()에 Array arr12(arr1);과 arr3 = arr1;과 같은 명령문을 추가하여 재정의된 연산자의 작업을 확인하시오. 복사구성자는 보관하는 배렬원소들이 자체의 기억기를 가지는 완전히 새로운 Array객체를 창조해야 한다. 복사구성자와 대입연산자들은 둘다 이전의 Array객체의 내용을 새로운 객체에 복사해야 한다. 어떤 크기의 Array를 다른 크기의 Array에 대입한다면 어떤 일이 생기는가?

5. 련습 1의 프로그람에서 bool형의 성원함수 IsOcerSize()를 Book와 Tape클라스에 추가하시오. 한권이 800페지이상인 책과 90분이상의 연주시간을 가지는 테프는 크기를 초과한다. main()으로부터 이 함수를 호출하고 도서와 테프의 초과크기를 취하는 문자렬 "크기초과"를 다른 자료와 함께 표시하시오. Book와 Tape객체들을 Publication형의 지적자배렬에 보관되는 지적자에 의하여 호출하려면 Publication기초클라스에 무엇을 추가해야 하는가? 이 기초클라스의 객체들을 실례화할수 있는가?

6. 화폐문자렬을 위한 재정의된 산수연산자 5개를 가지는 8장 련습 8의 프로그람을 고찰하시오. 거기에 재정의되지 않은 연산자를 두개 추가하시오. 조작

```
long double * bMoney
long double / bMoney
```

은 동료함수를 요구하므로 객체는 연산자의 오른변에, 수값상수는 왼변에 준다. main()함수에서 사용자가 둘째 화페문자렬과 류동소수점수값을 입력하고 그다음 적당한 값쌍들에 대하여 7개의 산수연산을 시험하시오.

7. 앞의 문제에서 9장 련습 8의 프로그람을 사용하시오. 이때 BMoney값을 가장 가까운 "원"으로 그리는 함수를 추가하시오. 그것을 다음과 같이 사용한다.

```
mo2 = Round(mo1);
```

이때 0.49원이하는 아래로 그리고 0.50원이상은 우로 그린다. modf1()서고함수를 사용하시오. 이 함수는 long double변수를 소수부와 옹근수부로 나눈다. 소수부가

0.50보다 작으면 그 옹근수부를 돌려주고 그렇지 않으면 1.0을 더한다. main()에서 49전이하로부터 50전이상으로 되는 BMoney량의 렬을 함수에 보내여 시험하시오.

8. 실례 10-25에서 단일자리수대신에 float형과 같은 실수에 대하여 식을 계산하게 하시오. 실례로

3.14159 / 7.0 + 75.25 * 3.3333 + 6.22

그러기 위하여 탄창을 더 발전시켜 두개의 연산자(char형)와 수(float형)을 보관할수 있다. 그런데 배렬에 기초하고있는 탄창에 어떻게 하면 두가지 다른 형을 보관할수 있겠는가?

char형과 float형은 크기가 다르다.

다른 형에로의 지적자를 보관할수 있는가?

그것들은 크기가 같지만 번역프로그람은 아직 한 배렬안에서 char*형과 float*형을 보관할수 없다. 두개의 다른 형의 지적자들을 같은 배렬에 보관하는 유일한 방법은 그것들을 같은 기초클라스로부터 파생시키는것이다. 한 클라스에는 char*를, 다른 클라스에는 float*를 보관할수 있으며 두개의 클라스를 기초클라스로부터 파생시킬수 있다. 그다음 두 종류의 지적자들을 기초클라스에로의 지적자배렬에 보관할수 있다. 기초클라스는 자체의 자료를 가질것을 요구하지 않는다. 즉 객체의 실례를 하나도 만들수 없는 추상클라스일수 있다. 구성자들은 일반적인 방법으로 파생클라스에 값들을 보관할수 있으나 순수가상함수를 사용하여 값들을 다시 꺼낼 필요는 없다. 여기에 가능한 실례들이 있다.

```
class Token
{
public:
   virtual float GetNumber()=0;
   virtual char GetOperator()=0;
};
class Operator : public Token
{
private:
   char oper; // 연산자 +, -, *, /
public:
   Operator(char);
   char GetOperator();
   float GetNumber();
};
class Number : public Token
{
private:
  float fnum;
public:
   Number(float);
   float GetNumber();
```

```
    char GetOperator();
};
Token* aToken[100];
```

기초클라스의 가상함수들은 모든 파생클라스들에서 실례화되지 않으면 파생클라스는 자체로 추상으로 된다. 따라서 Operator클라스는 수를 보관하지 않지만 GetNumber()함수를 요구하며 Number클라스는 연산자를 보관하지 않지만 GetOperator()함수를 요구한다. 이 틀거리를 프로그람으로 전개하시오. Token객체들을 보관하는 Stack클라스를 추가하시오. main()에서 여러개의 연산자(+와 * 등)와 류동소수점수(1.12 등)을 탄창에 넣거나 꺼내보시오.

9. 1장 련습 4에서는 LinkList클라스에 재정의된 해체자를 추가하였다. 이렇게 해체자를 확장한 클라스의 기정대입연산자에 기초하여 list2 = list1;와 같은 명령문을 사용해서 전체 클라스를 대입한다. 그러면 후에 list1객체를 삭제할수 있다. list2에서 아직도 같은 자료를 호출할수 있는가? 호출할수 없다.

list1이 삭제될 때 그 해체자는 모든 련결을 삭제하기때문이다. list2객체에 실제로 포함되여있는 유일한 자료는 첫 련결에로의 지적자뿐이다. 따라서 list2의 지적자는 무효로 되므로 목록의 호출은 무의미한 값을 유도하게 되며 프로그람은 오유를 일으킨다. 이것을 방지하는 하나의 방법은 대입연산자를 재정의하여 LinkList객체자체는 물론 자료련결을 모두 복사하는것이다. 사슬을 따라가면서 매개 련결을 차례로 복사해야 한다. 물론 복사구성자도 재정의해야 한다. main()에서 지적자와 new를 사용하여 LinkList객체들을 실제로 창조한다. 모든 자료의 복사는 기억기사용의 견지에서 그리 효과가 없다. 이 방법을 실례 11-18에서 사용한 수법과 비교하시오.

10. 련습 7에서 론의한 변경을 실례 10-25에 대하여 진행하시오. 즉 류동소수점수를 포함하는 식을 해석하도록 하시오. 련습 7의 클라스들과 실례 10-25의 알고리듬을 결합하시오. 문자들대신에 Token에로의 지적자에 대하여 조작해야 한다. 이것은 다음과 같은 명령문을 포함한다. 즉

```
Number* ptrN = new Number(ans);
s.Pusj(ptrN);
```

과

```
Operator* ptrO = new Operator(ch);
s.Push(ptr0);
```

제 12 장. 스트림과 파일

이 장에서는 C++의 스트림클라스들의 계층구조를 고찰하고 그 특성을 요약한다.

이 장에서 중요한것은 C++스트림을 사용하여 파일관련조작을 처리하는 방법을 보여주는데 있다. 여러가지 방법으로 파일에 자료를 읽고 쓰는 방법과 오유조종방법, 파일과 객체지향프로그람작성법과의 관계를 보여준다. 기억기내에서 본문형식화, 지령행인수, 삽입연산자와 발취연산자의 재정의, 인쇄기에로의 자료전송 등 파일과 관련한 C++의 여러가지 다른 특성도 설명한다.

제 1 절. 스트림클라스

스트림(stream)은 자료의 흐름(flow)에 주어진 일반적인 이름이다.

C++에서 스트림은 특정한 클라스의 객체로 표시된다.

지금까지 우리는 cin과 cout스트림클라스의 객체들을 사용하였다.

여기서 각이한 종류의 자료흐름을 표시하는데 여러가지 스트림을 사용한다.

실례로 ifstream클라스는 입력디스크파일로부터의 자료흐름을 표시한다.

1. 스트림의 우점

C프로그람작성자는 printf()와 scanf, fprintf()와 fscanf()와 같은 전통적인 C함수대신에 스트림클라스를 입출력에 사용하는것이 어떠한 우점을 가지는가에 대하여 의심할수 있다.

그 하나의 리유는 단순성에 있다. printf()에서 %f와 %d형식문자를 사용해보면 이것을 알수 있다. 스트림의 매개 객체는 이미 자기를 표시하는 방법을 알고있으므로 그러한 형식문자가 없다. 이것은 오유의 기본원천을 제거한다.

또 하나의 리유는 삽입과 발취연산자와 같은 현존연산자와 함수들을 재정의하여 사용자가 창조한 클라스에 대하여 작업할수 있는데 있다. 이것은 사용자의 클라스들이 기본형과 같은 방법으로 동작하게 하고 프로그람작성을 더 쉽게 그리고 오유가 없게 한다. Windows와 같이 화면에로의 직접본문출력을 사용하지 않는 도형방식사용자대면부의 환경에서 프로그람을 작성할 계획이라면 여기서도 스트림입출력이 중요한가에 대하여 의심할수 있다.

그래도 C++스트림에 대하여 알아야 하는가? 그렇다.

그것은 파일에 자료를 써넣는 최신의 방법이고 또한 본문입출력창문들과 다른 도형방식사용자대면부의 요소하에서 후에 사용하기 위해 기억기안에 자료를 형식화하는 발전된 방법이기때문이다.

2. 스트림클라스계층

스트림클라스는 좀 복잡한 계층으로 배렬되여있다. 그림 12-1은 가장 중요한 클라스들의 배렬을 보여준다.

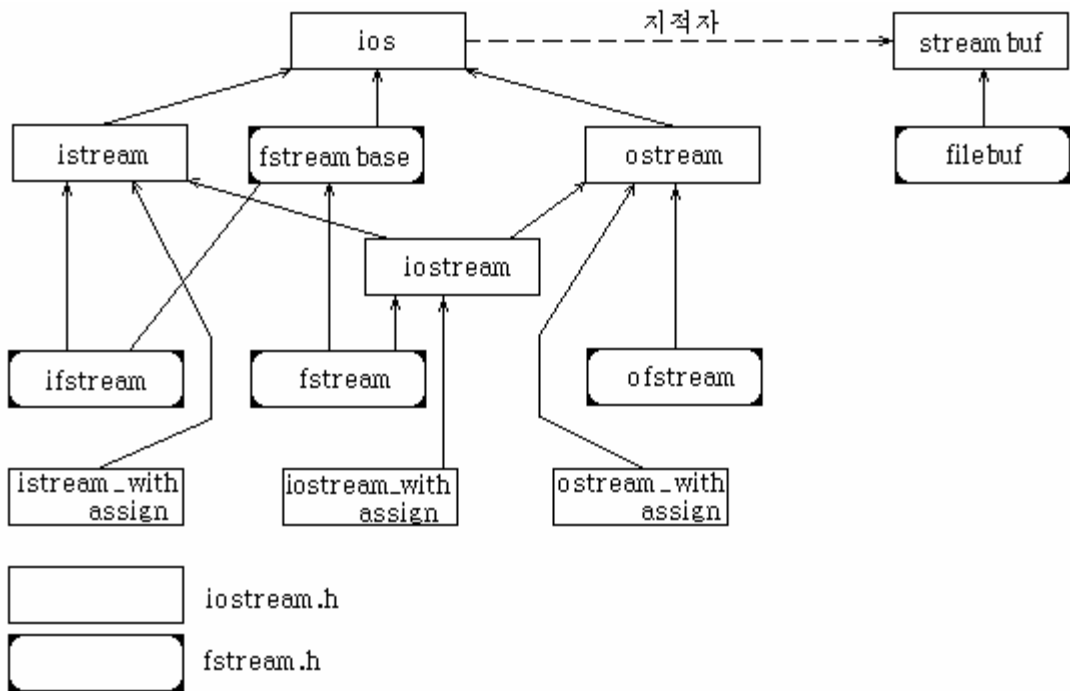



그림 12-1. 스트림클라스계층

이미 일부 스트림클라스를 사용하였다. 발취연산자 >>는 istream클라스의 성원이고 삽입연산자 <<는 ostream클라스의 성원이다. 이 두 클라스는 ios클라스로부터 파생된다. 오유출력스트림을 표시하는 cout객체는 보통 영상표시장치를 가리키고 ostream클라스에서 파생된 ostream_withassign클라스의 미리 정의된 객체이다.

영상표시장치와 건반으로부터의 입출력에 사용하는 클라스들은 앞장의 실례들에서 포함시킨 머리부파일 IOSTREAM에서 선언된다. 디스크파일입출력에 사용되는 클라스들은 FSTREAM파일에서 선언된다. 그림 12-1은 두개의 머리부파일에 어떤 클라스들이 있는가를 보여준다.(또한 일부 조작자는 IOMANIP에서 선언되고 기억기조작 클라스들은 STRSTREAM에서 선언된다.) 이 머리부파일들을 표시하여 여러가지 클라스들사이의 관계를 추적할수 있다. 머리부파일들은 번역프로그람의 INCLUDE등록부에 있다.

스트림에 대한 몇가지 질문에 대한 대답은 그 클라스의 상수성원들을 고찰하여 얻을수 있다. 그림 12-1에서 알수 있는것처럼 ios클라스는 계층구조의 기초클라스이다. ios는 모든 종류의 입출력조작에서 공통적으로 쓰이는 많은 상수와 성원함수들을 포함하고있다. showpoint와 fixed형식기발과 같은 일부 기능은 이미 사용하였다. 또한 ios클라스는 자료를 읽거나 쓰는 실제기억완충기를 포함하는 streambuf클라스에로의 지

적자와 이 자료를 조종하는 저수준프로그람을 포함한다. 보통 streambuf클라스는 다른 클라스에 의해 자동적으로 참고되므로 이에 대하여 걱정할 필요는 없다.

istream과 ostream클라스들은 ios로부터 파생되고 각각 입력과 출력에 사용된다. istream클라스는 get(), getline(), read(), 재정의된 발취연산자(>>)를 포함하고 ostream은 put(), write(), 재정의된 삽입연산자(<<) 등을 포함한다.

iostream클라스는 istream과 ostream으로부터 다중계승에 의해 파생된다. 이 클라스로부터 파생된 클라스들은 디스크파일과 같이 장치와 함께 사용할수 있으며 입출력에 동시에 사용할수 있다. 세개의 클라스 istream_withassign, ostream_withassign, iostream_withassign은 각각 istream, ostream, iostream에서 파생된다. 이 세개의 클라스들은 istream, ostream, iostream스트림클라스들에 대입연산자를 추가한다. 스트림클라스의 이러한 요약을 추상적으로 볼수 있다.

3. ios클라스

ios클라스는 모든 스트림클라스의 선조로서 C++스트림을 조작하는데 필요한 주요 루틴들을 포함한다. 가장 중요한 세가지 특성은 형식기발, 오유상태기발, 파일조작방식이다.

1) 형식기발

형식기발(formatting flag)은 ios의 enum정의의 모임이다. 형식기발은 입출력형식과 조작의 여러가지 방식을 선택하는 on/off여닫이로서 동작한다. 이미 이것들을 사용하고있으므로 매개에 대하여 구체적으로 설명하지 않는다. 표 12-1은 형식기발의 완전한 목록을 보여준다.

형식기발을 설정하는 여러가지 방법이 있으며 매개의 기발을 여러가지 방법으로 설정할수 있다. 그것들은 ios클라스의 성원이므로 앞에 ios와 범위해결연산자를 써야한다. (실례로 ios::skipws)

표 12-1. ios형식기발

기발	의 미
skipws	입력할 때 공백을 무시한다.
left	왼쪽으로 맞추어 출력한다.[12.34]
right	오른쪽으로 맞추어 출력한다.[12.34]
interval	부호 혹은 기수지시자와 수자사이의 공간을 사용한다. [+ 12.34]
dec	10진수로 변환한다.
oct	8진수로 변환한다.
hex	16진수로 변환한다.
boolalpha	bool을 "true" 혹은 "false"문자렬로 변환한다.

기발	의 미
showbase	출력시 기수지시자를 표시한다.(8진수일 때 0, 16진수일 때 0x)
showpoint	출력할 때 소수점을 표시한다.
uppercase	대문자 X,E와 16진출력문자(ABCDEF)를 사용한다. 기정은 소문자.
showpos	정의 옹근수앞에 +를 표시한다.
scientific	류점출력할 때 지수형식을 사용한다.[9.1234E2]
Fixed	류점출력할 때 고정소수형식을 사용한다.[912.34]
unitbuf	삽입한 후 모든 스트림클라스들을 초기화한다.
Stdio	삽입후에 stdout, stderr를 초기화한다.

모든 기발은 ios의 성원함수 setf()와 unsetf()에 의하여 설정한다. 례를 들면

```
cout.setf(ios::left);
cout << "This text is left-justified";
cout.unsetf(ios::left);  // 기정으로 복귀(오른쪽 맞추기)
```

대부분의 형식기발은 조작자에 의하여 설정할수 있다.

2) 조작자

조작자(manipulater)는 스트림에 직접 삽입할수 있는 형식지령이다. 조작자 endl과 같은 실례를 이미 보았다. endl은 스트림에 행바꾸기문자를 보내고 그 내용을 출력하고 스트림을 초기화한다.

```
cout << "To each his own" << endl;
```

또한 setiosflags()조작자를 사용하였다.(실례 7-4)

```
cout << setiosflags(ios::fixed)          // 고정소수점수형식
     << setiosflags(ios::showpoint)     // 항상 표시
     << var;
```

실례들이 보여주는것처럼 조작자에는 두가지 즉 인수를 가지는것과 인수를 가지지 않는것이 있다. 표 12-2에는 인수없는 주요 조작자들을 보여준다.

표 12-2. 인수없는 ios조작자

기발	의 미
ws	입력할 때 공백을 무시한다.
dec	10진수로 변환한다.
oct	8진수로 변환한다.
hex	16진수로 변환한다.
endl	행바꾸기를 삽입하여 출력하고 출력스트림을 초기화한다.
ends	출력문자렬을 완료하기 위하여 null문자를 삽입한다.
flush	출력스트림을 초기화한다.
lock	파일핸들을 쇄정한다.
unlock	파일핸들의 쇄정을 해제한다.

스트림에 이 조작자들을 직접 삽입할수 있다. 실례로 var를 16진형식으로 출력하려면

```
cout << hex << var;
```

조작자는 스트림에서 그 앞에 오는 자료가 아니라 뒤에 오는 자료에만 영향을 준다. 표 12-3은 인수있는 주요 조작자들을 보여준다. 이 함수들에는 IOMANIP머리부파일이 필요하다.

표 12-3. 인수있는 ios조작자

조작자	인 수	목 적
setw()	마당폭(int)	출력용마당폭을 설정한다.
setfill()	채움문자(int)	출력용채움문자를 설정한다.(기정은 공백)
Setprecision()	정확도(int)	정확도(표시자리수)를 설정한다.
setiosflags()	형식기발(long)	지정된 기발을 설정한다.
resetiosflags()	형식기발(long)	지정된 기발을 해제한다.

3) 함수

ios클라스는 형식기발을 설정하여 다른 일감들을 처리하는데 사용할수 있는 함수들을 많이 가지고있다. 표 12-4는 오유취급을 제외한 대부분의 함수를 보여준다.

표 12-4. ios함수

함수	목적
ch=fill();	채움문자를 돌려준다.(마당의 비여있는 부분을 채운다. 기정으로는 공백이다.)
fill(ch);	채움문자를 설정한다.
p=precision();	정확도(류동소수점수의 자리수)를 얻는다.
precision(p);	정확도를 설정한다.
w=width();	현재 마당폭(문자수)을 얻는다.
width(w);	현재 마당폭을 설정한다..
setf(flags);	지정된 형식문자를 설정한다.(실례로 ios::left)
unsetf(flags);	지정된 형식기발을 해제한다.
setf(flags,field);	먼저 마당을 지우고 기발을 설정한다.

이 함수들은 표준점연산자에 의하여 특정한 스트림객체에 대하여 호출한다. 실례로 마당폭을 14로 설정하려면 명령문을 다음과 같이 쓴다.

```
cout.width(14);
```

다음의 명령문은 채움문자를 별표로 설정한다.

```
cout.fill('x');
```

여러개의 함수를 사용하여 ios형식기발을 직접 조작할수 있다. 실례로 왼쪽맞추기를 설정하면

cout.setf(ios::left);

오른쪽맞추기를 사용하려면 명령문을 다음과 같이 쓴다.

cout.unsetf(ios::right);

setf()의 2인수판은 두개의 인수를 사용하여 특별한 형 또는 마당의 모든 기발들을 재설정한다. 그다음 첫 인수로서 지정된 기발을 설정한다. 이것은 련관된 기발들을 쉽게 재설정하게 한다. 표 12-5에서 그 배치를 보여준다.

표 12-5. setf()의 2인수판

조작자	인 수
dec, oct, hex	basefield
left,right,interval	adjustfield
scientific,fixed	floatfield

실례로

cout.setf(ios::left, ios::adjustfield);

는 본문조작을 취급하는 모든 기발을 지우고 왼쪽맞추기출력을 위한 left기발을 설정한다.

형식기발을 여기에 준 방법으로 사용함으로써 건반과 영상표시장치뿐아니라 파일에 대한 입출력을 형식화할수 있다.

4. isteam클라스

ios로부터 파생된 istream클라스는 입력의 고유한 동작 혹은 발취를 처리한다.(표 12-6)

발취는 출력조작 즉 삽입과 혼돈하기 쉽다. 그림 12-2는 그 차이를 강조한다.

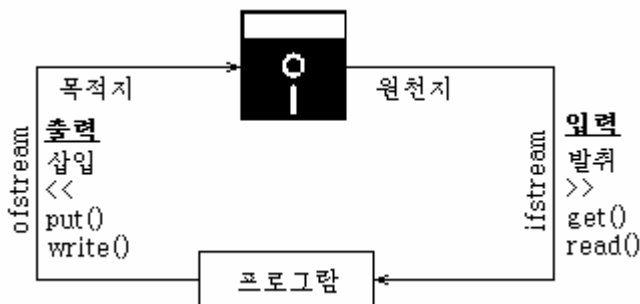



그림 12-2. 파일입출력

표 12-6. istream함수

함 수	목 적
>>	모든 기본형과 재정의된 형을 위한 형식화된 발취성원함수
get(ch)	한개 문자를 ch에 꺼낸다.

함　수	목　적
get(str)	배렬 str에로 0일 때까지 문자들을 꺼낸다.
get(str,MAX)	배렬 str에로 MAX개의 문자를 꺼낸다.
get(str,DELIM)	지정된 구분기호가 나올 때까지 배렬 str에서 문자를 꺼낸다.(대체로 '\n') 스트림에 구분문자를 남긴다.
get(str,MAX,DELIM)	출력문자렬을 완료하기 위하여 null문자를 삽입한다. 스트림에 구분문자를 남긴다.
getline(str,MAX,DELIM)	배렬 str에 MAX개 문자 혹은 DELIM문자까지 문자를 꺼낸다. 구분문자를 꺼낸다.
putback(ch)	입력스트림에 마지막으로 읽은 문자를 삽입한다.
ignore(MAX,DELIM)	지정된 구분기호까지 포함하여 MAX문자까지 꺼내서 무시한다.
peek(ch)	한개 문자를 읽고 그것을 스트림에 남긴다.
cout=gcount()	get(),getline(),read()의 직접호출에 의해 읽어들인 문자수를 돌려준다.
read(str,MAX)	파일에 대하여 str에 MAX개 문자까지, EOF까지 꺼낸다.
seekg()	파일의 선두로부터 파일지적자의 변위(byte수)를 설정한다.
seekg(pos,seek_dir)	파일안의 지정된 위치로부터 파일지적자의 위치(byte수)를 설정한다. seek_dir는 ios::beg,ios::cur,ios::end일수 있다.
pos=tellg(pos)	파일의 선두로부터 파일지적자의 위치(byte수)를 돌려준다.

여기서 get()와 같은 일부 함수를 이미 보았다. 대부분은 건반으로부터의 자료흐름을 표시하는 cin객체에 대하여 동작한다. 그러나 마지막 4개는 디스크파일과 작업한다.

5. ostream클라스

ostream클라스는 출력 혹은 삽입조작을 조종한다. 표 12-7은 가장 일반적으로 쓰이는 성원함수들을 보여준다. 표에서 마지막 4개 함수는 디스크파일에 대하여 동작한다.

표 12-7.　　　　　　　　　　　ostream함수

함　수	목　적
<<	모든 기본형과 재정의된 형을 위한 형식화된 삽입성원함수
put(ch)	스트림에 문자 ch를 삽입한다.
flush()	완충기의 내용을 모두 내보내고 행바꾸기를 삽입한다.
write(str,SIZE)	파일에 배렬 str의 SIZE개의 문자를 삽입한다.
seekp(position)	파일의 선두로부터 파일지적자의 변위(byte수)를 설정한다.
seekp(position,seek_dir)	파일안의 지정된 위치로부터 파일지적자의 변위(byte수)를 설정한다. seek_dir는 ios::beg,ios::cur,ios::end일수 있다.

함 수	목 적
pos = tellp()	파일지적자의 위치(byte수)를 돌려준다.

6. iostream과 _withassign클라스들

두개의 istream과 ostream으로부터 파생되는 iostream클라스는 다른 클라스(특히 iostream_withassign)들을 파생시키는 기초클라스로서만 동작한다. 여기에는 구성자와 해체자를 제외한 다른 함수가 없다. iostream으로부터 파생된 클라스들은 입출력을 모두 처리할수 있다.

아래에 세개의 _withassign클라스가 있다.

· istream에서 파생된 istream_withassign

· ostream에서 파생된 ostream_withassign

· iostream에서 파생된 iostream_withassign

_withassign클라스들은 재정의된 대입연산자를 포함하며 그 객체들을 복사할수 있는것을 제외하면 기초클라스와 같다.

그러면 왜 복사가능한 스트림과 복사불가능한 스트림클라스를 구분하는가?

일반적으로 스트림클라스의 객체들을 복사하는것은 좋지 않다. 그 리유는 매개의 객체가 객체의 실제자료를 보관하기 위하여 기억기의 어떤 령역을 차지하고있는 특별한 streambuf객체와 련결되기때문이다. 스트림객체를 복사하면 streambuf객체도 복사되므로 혼란이 일어난다. 그러나 일부 경우에는 스트림을 복사하는것이 중요하다.

따라서 istream, ostream, iostream클라스들은 재정의된 대입연산자와 복사구성자를 비공개로 하여 복사할수 없게 만들었고 그로부터 파생된 _withassign클라스를 복사할수 있게 하였다.

- 미리 정의된 스트림객체

이미 _withassign클라스로부터 파생되는 두개의 미리 정의된 스트림객체 cin과 cout를 많이 사용하였다. 이것들은 보통 건반과 영상표시장치와 련결된다.

또한 두개의 미리 정의된 다른 객체 cerr와 clog가 있다.

· cin은 istream_withassign의 객체이고 보통 건반입력에 사용된다.

· cout은 ostream_withassign의 객체이고 영상표시장치에 사용된다.

· cerr은 ostream_withassign의 객체이고 오유통보에 사용된다.

· clog은 ostream_withassign의 객체이고 log통보에 사용된다.

cerr객체는 오유통보와 프로그람진단에 쓰인다. cerr에 보낸 출력은 cout와 달리 완충되지 않고 즉시 표시된다. 또한 후에 반영(redirect)되지 않는다. 이러한 리유로 작성자의 프로그람이 너무 이르게 끝나면 cerr로부터 마지막 출력통보문을 볼수 있는 더 좋은 기회를 얻을수 있다.

다른 객체 clog는 후에 반영된다는데서 cerr와 비슷하지만 출력은 완충된다. 그러

나 cerr의 출력은 완충되지 않는다.

제 2 절. 스트림오유

지금까지는 cout << "Good morning";와 cin >> var;형식의 명령문을 사용하여 직선적인 입출력방법을 많이 사용해왔다.

그러나 이 수법은 입출력처리과정에 오유가 없다는것을 가정한다. 그러나 항상 그런것은 아니다.

특히 입력할 때에 수자 9대신에 사용자가 문자렬 "nine"을 입력하거나 아무것도 입력하지 않고 Enter건을 누르면 어떤 일이 생기는가?

또한 하드웨어실패가 있으면 어떤 일이 생기는가?

이 절에서는 오유가 발생하는 경우를 고찰한다.

1. 오유상태기발

스트림오유상태기발은 입력이나 출력조작에서 발생한 오유를 알려주는 ios렬거성원이다.(표 12-8) 그림 12-3은 이 기발들을 보여준다. 여기서 ios함수는 표 12-9와 같이 오유기발을 읽는데 사용할수 있다.

표 12-8. 오유상태기발

이름	의 미
goodbit	오유없다. (기발이 설정되지 않았다. 값은 0)
eofbit	파일끝에 이르렀다.
failbit	조작이 실패하였다. (사용자오유, 너무 이른 EOF)
badbit	무효한 조작(streambuf와 련결되지 않음)
hardbit	회복불가능한 오유

표 12-9. 오유기발에 대한 함수

함수	목 적
int=good()	모든것이 좋으면 즉 기발이 설정되지 않았으면 true를 돌려준다.
int=eof()	EOF기발이 설정되였으면 true를 돌려준다.
int=fail()	failbit, badbit 혹은 hardbit기발이 설정되였으면 true를 돌려준다.
int=bad()	badbit 혹은 hardbit기발이 설정되였으면 true를 돌려준다.
clear(int=0)	인수가 없으면 모든 오유비트들을 삭제하고 인수가 있으면 지정된 기발을 설정한다. (ios::failbit)

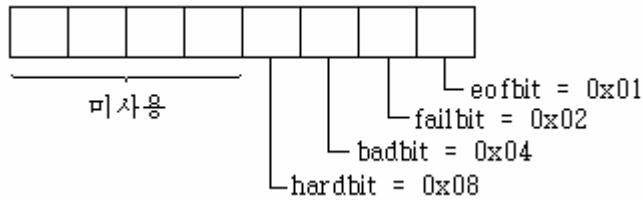



그림 12-3. 스트림상태기발

2. 수값의 입력

수값을 입력할 때 오유조종방법을 고찰하자. 이 방법은 goodbit의 값을 검사하는 것인데 건반과 파일로부터 수값을 읽어들일 때 적용한다. goodbit는 사용자에게 정확히 입력할수 있는 또 한번의 기회를 준다.

```
while(true)
{
   cout << "\n옹근수를 입력하시오:";
   cin >> i;
   if(cin.good())                    // 오유가 없으면
   {
      cin.ignore(10, '\n');      // 행바꾸기를 제거한다.
      break;                          // 순환완료
   }
   cin.clear();                      // 오유비트지우기
   cout << "부정확한 입력";
   cin.ignore(10, '\n');         // 행바꾸기를 제거한다.
}
cout << "옹근수는 " << i;         // 오유없는 옹근수
```

건반입력을 읽어들일 때 이 프로그람이 탐지하는 가장 일반적인 오유는 사용자가 비수값(9대신에 "nine")을 입력하는것이다. 이때 failbit가 설정된다. 이것은 디스크파일에서 일반적인 체계관련실패를 탐색할수 있게 한다.

류동소수점수(float, double, long double)는 옹근수와 같은 방법으로 오유를 해석한다.

3. 너무 많은 문자의 입력에 대한 처리

너무 많은 문자를 입력스트림으로부터 읽어들일 때 문제를 일으킬수 있다.

이것은 오유가 있을 때 특히 더하다. 일반적으로 입력이 끝난 후에 입력스트림에는 여유문자들이 남아있다. 그리고 다음의 입력조작에로 넘어간다. 비록 그런 일이 없다해도 흔히 끝에 행바꾸기 혹은 다른 문자들이 남아있는 경우가 있다.

여유문자들을 제거하기 위하여 istream의 ignore(MAX, DELIM)성원함수를 사용할수 있다. ignore()는 지정된 구분문자를 포함하여 MAX개 문자까지 읽어들인다.

실례에서 다음행

```
cin.ignore(10, '\n');
```

은 cin이 '\n'을 포함하여 10개의 문자까지만 읽어들이고 나머지는 입력에서 삭제한다.

4. 입력없는 입력

일반적으로 타브, 공백, '\n'과 같은 문자들은 수를 입력할 때 무시된다. 이것은 부작용을 일으킬수 있다. 실례로 앞에 보여준 코드에서 단순명령문

```
cin >> i;
```

와 같이 사용자가 수를 입력할데 대한 재촉문에 어떤 수자도 입력하지 않고 Enter건을 누를수 있다. Enter를 누르면 유표가 다음 행에 내려가고 스트림은 수를 입력하기를 기다린다.

그러면 유표가 다음 행으로 내려갈 때 어떤 일이 생기는가?

우선 경험없는 사용자는 Enter건을 누를 때 프로그람이 파괴되는것으로 생각할수 있다.

다음으로 Enter건을 계속 반복하여 누르면 보통 전체화면이 우로 흘러가기 시작할 때까지 유표가 아래로 내려간다. 이것은 타자기형식의 입력에서 흔히 볼수 있다. 그러나 본문에 기초하는 도형그리기프로그람에서 화면의 흘림은 재배렬을 가져오고 우연히 화면은 지워진다.

따라서 입력스트림에 공백을 무시하지 않는다고 알리는것이 중요하다. 이것은 skipws기발을 리용하여 조종할수 있다.

```
cout << "\n옹근수를 입력하시오:";
cin.unsetf(ios::skipws);              // 공백을 무시하지 않는다.
cin >> i;
if(cin.good())
{
   // 오유없다.
}
// 오유
```

이제 사용자가 수를 입력하지 않고 Enter건을 누르면 failbit가 설정되고 오유가 발생한다. 그다음 프로그람은 사용자에게 무엇을 하겠는가를 묻고 유표를 재배치하여 화면이 흘러가지 않게 한다.

5. 문자렬과 문자의 입력

실제로 문자렬과 문자를 입력하는 경우에는 오유가 발생하지 않으므로 모든 입력을 문자렬로 해석할수 있다. 디스크파일로부터 입력하면 문자와 문자렬은 여전히 오유검사되고 EOF 혹은 어떤 잘못된 경우와 만나게 된다. 수값을 입력할 때와는 달리 문자렬과 문자를 입력할 때에는 공백을 무시한다.

6. 오유없는 Distance클라스

사용자가 Distance클라스에 입력하는 프로그람에서 오유검사방법을 고찰하자. 이 프로그람은 단순히 사용자로부터 메터와 센치메터로 된 Distance를 받아들인다. 그러나 사용자가 입력오유를 발생시키면 프로그람은 사용자에게 적당한 설명을 표시하고 다시 입력하기를 기다린다.

프로그람에서는 앞에서 설명한 수법에 의하여 오유를 조종하기 위한 성원함수 GetDist()를 확장하였다. 또한 사용자가 메터에 류동소수점수를 입력하지 않는다는것을 담보하기 위하여 여러개의 명령문을 삽입하였다. 이것은 메터값이 옹근수이고 센치메터는 류동소수점수이므로 사용자가 혼돈하지 않도록 하기 위한것이다.

보통 옹근수를 요구하는 경우에 소수점이 나타나면 발취연산자는 오유를 경고하지 않고 완료한다. 그러한 오유를 알아야 할 때 메터값을 int대신 문자렬로 읽을수 있다. 그다음 문자렬이 메터의 정확한 값이면 true를 돌려주는 함수 IsMeter()를 사용하여 문자렬을 검사한다. 메터검사에서 통과하려면 오직 수자들을 포함해야 하며 -999～999사이의 수로 평가되여야 한다. (Distance클라스를 매우 큰 메터값을 측정하는데 사용하지 않는다고 가정한다.) 문자렬이 메터검사에서 통과되면 그것을 서고함수 atoi()를 사용하여 int로 변환한다.

센치메터값은 류동소수점수이므로 그 범위 즉 0보다 크거나 100.0보다 작은가를 검사한다. 또한 ios오유기발을 검사한다. 대체로 failbit는 사용자가 수자가 아닌 기호를 입력하였을 때 설정된다. 여기에 실례 12-1의 프로그람이 있다.

(실례 12-1) Distance클라스에서 입력검사

```
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;
int IsMeter(string);
class Distance
{
private:
   int meters;
   float centies;
public:
   Distance() : meters(0), centies(0.0) {}
   Distance(int me, float ce) { meters = me; centies = ce; }
   void ShowDist() { cout << meters << "m " << centies << "cm"; }
   void GetDist();
};
void Distance::GetDist()
{
   string instr;
   while(true) {
```

```
        cout << "\n\n메터를 입력하시오: ";
        cin.unsetf(ios::skipws);
        cin >> instr;
        if(IsMeter(instr)) {
            cin.ignore(10, '\n');
            meters = atoi(instr.c_str());
            break;
        }
        cin.ignore(10, '\n');
        cout << "메터는 1000이하의 옹근수이여야 합니다.\n";
    }
    while(true)
    {
        cout << "센치메터를 입력하시오: ";
        cin.unsetf(ios::skipws);
        cin >> centies;
        if(centies >= 100.0 || centies < 0.0)
        {
            cout << "센치메터는 0.0과 99.99값사이이여야 합니다.\n";
            cin.clear(ios::failbit);
        }
        if(cin.good())
        {
            cin.ignore(10, '\n');
            break;
        }
        cin.clear();
        cin.ignore(10, '\n');
        cout << "부정확한 센치메터를 입력하였습니다.\n";
    }
}
int IsMeter(string str)
{
    int sLen = str.size();
    if(sLen == 0 || sLen > 5)
        return 0;
    for(int j=0; j<sLen; j++)
    {
        if((str[j] < '0' || str[j] > '9') && str[j] != '-')
            return 0;
    }
    double n = atof(str.c_str());
    if(n < -999.0 || n > 999.0)
        return 0;
    return 1;
}
int main()
{
```

```
    Distance d;
    char ans;
    do
    {
        d.GetDist();
        cout << "\nDistance="; d.ShowDist();
        cout << "\n계속하겠습니까(y/n)?"; cin >> ans;
        cin.ignore(10, '\n');
    } while(ans != 'n');
    return 0;
}
```

여기서는 수동적인 방법으로 오유상태기발을 설정한다. 즉 센치메터값이 0보다 크거나 100.0보다 작다는것을 담보하려고 한다. 그렇지 않으면 명령문

```
cin.clear(ios::failbit); // failbit를 설정한다.
```

을 사용하여 failbit를 설정한다.

cin.good()에 의해 오유를 검사하고 failbit가 설정되였으면 입력이 옳지 않다고 경고한다.

제 3 절. 스트림에 의한 디스크파일입출력

대부분의 프로그람은 디스크파일에 자료를 보관하고 그로부터 읽어들여야 한다.

디스크파일과의 작업은 다른 일련의 클라스 즉 입력에는 ifstream, 입출력에는 fstream, 출력에는 ofstream을 요구한다. 이 클라스들의 객체는 디스크입출력과 관련되여있으며 성원함수들에 의하여 파일로부터 읽거나 파일에 써넣을수 있다.

그림 12-1을 다시 참고하면 ifstream은 istream으로부터, ofstream은 ostream으로부터 파생되는것을 알수 있다. 또한 그 부모클라스들은 ios에서 파생된다. 이리하여 파일지향의 클라스들은 일반적인 클라스들로부터 더 많은 성원함수들을 물려받는다.

또한 파일지향의 클라스들은 fstreambase클라스로부터 다중계승에 의해 파생된다. fstreambase클라스는 파일지향의 완충기인 filebuf클라스의 객체를 포함하며 더 일반적인 streambuf클라스로부터 계승된 성원함수들도 포함한다. 그러나 이 완충기클라스에 대하여 걱정할 필요는 없다.

ifstream, ofstream, fstream클라스들은 FSTREAM파일에서 선언된다.

C프로그람작성자들은 C++에서 사용하는 디스크입출력수법이 C와 다르다는것을 알수 있다. fread(), fwrite()와 같은 이전의 C함수들은 C++에서 여전히 작업한다. 그러나 C함수들은 객체지향환경에 적합하지 않다. 새로운 C++수법은 아주 명백하고 실현하기 쉽다.(낡은 C함수들과 C++스트림을 같이 사용해서는 안된다. 그것들이 서로 협동하게 하는 방도가 있다고 해도 잘 동작하지 않는다.)

1. 형식화된 파일입출력

형식화된 입출력에서 수값은 문자들의 렬로서 디스크에 보관된다. 따라서 6.02는 4byte float형 또는 8byte double형으로 보관되지 않고 문자 '6', '.', '0', '2'로 보관된다. 이것은 자리수가 많은 수값들에는 비효과적이지만 많은 경우에 적합하고 실현하기 쉽다.

1) 자료의 써넣기

실례 12-2는 디스크파일에 문자 하나, 옹근수 하나, double형 하나, string객체 두개를 써넣으며 화면에는 출력하지 않는다.

(실례 12-2) <<에 의하여 파일에로의 형식화된 출력

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main()
{
  char ch = 'x';
  int j = 77;
  double d = 6.02;
  string str1 = "Kafha";
  string str2 = "prowt";
  ofstream outFile("FData.txt");
  outFile << ch << j << ' ' << d << str1 << ' ' << str2;
  cout << "파일에 출력하였습니다.\n";
  return 0;
}
```

여기서는 outFile을 ofstream클라스의 객체로 정의한다. 동시에 그것을 파일 fdata.txt로 초기화한다. 초기화는 파일을 위한 여러가지 속성을 설정하고 디스크에 있는 파일을 호출하거나 열어놓는다. 파일이 존재하지 않으면 그것을 새로 창조한다. 파일이 존재하면 그것을 삭제하고 새로운 자료로 낡은 자료를 교체한다. outFile객체는 이전 프로그람의 cout처럼 동작하므로 파일에 기본형의 변수를 출력할 때 삽입연산자(<<)를 사용할수 있다. 이것은 ofstream의 기초클라스 ostream에서 삽입연산자가 적당히 재정의되여있기때문이다.

프로그람을 완료할 때 outFile객체는 범위밖으로 벗어난다. 그러면 해체자가 호출되고 파일이 자동적으로 닫기므로 파일을 명시적으로 닫지 않아도 된다.

몇가지 문제가 있다.

첫째로, 수값을 비수값문자와 분리해야 한다. 수값이 고정길이마당이 아니라 문자들의 렬로 보관되므로 이것은 파일로부터 자료를 읽어들일 때 어떤 수값이 끝나고 다른 수값이 시작되는가를 삽입연산자가 알아내는 유일한 방법이다.

둘째로, 같은 리유에 의하여 문자렬들을 공백으로 구분해야 한다. 이것은 문자렬안

에 공백이 포함될수 없다는것을 암시한다. 이 실례에서는 두개의 구분문자에 대하여 공백문자(' ')를 사용한다. 문자는 고정길이를 가지므로 구분문자를 요구하지 않는다.

실례 12-2가 Fdata.txt파일에 자료를 써넣었다면 Windows의 WordPad나 DOS지령 TYPE로 검사할수 있다.

2) 자료의 읽기

파일이름으로 초기화된 ifstream객체를 사용하여 실례 12-2에서 생성한 파일을 읽을수 있다. 그 파일은 객체가 창조될 때 자동적으로 열린다. 그다음 발취(>>)연산자에 의하여 파일로부터 읽을수 있다.

여기에 Fdata.txt파일로부터 자료를 읽어들이는 실례 12-3이 있다.

(실례 12-3) >>에 의하여 파일로부터 형식화된 출력의 읽어들이기

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main()
{
   char ch;
   int j;
   double d;
   string str1, str2;
   ifstream inFile("FData.txt");
   inFile >> ch >> j >> d >> str1 >> str2;
   cout << ch << endl << j << endl << d << endl << str1 << endl << str2<<endl;
   return 0;
}
```

inFile이라고 이름지은 ifstream객체에 대하여 이전 프로그람들에서의 cin처럼 동작한다. 즉 파일에 자료를 삽입한다. 자료를 정확히 형식화하면 그것을 발취할 때 오유가 없으며 적당한 변수에 그것을 보관하고 그 내용을 표시한다.

프로그람의 출력은 다음과 같다.

```
x
77
6.02
Kafha
prowt
```

물론 수들은 프로그람에서 사용하는 2진표시로 변환된다. 즉 77은 j변수에 문자가 아니라 int형으로 보관하고 6.02는 double로 보관한다.

2. 공백을 포함하는 문자렬

마지막 실례는 파일을 포함하며 char*문자렬에서 동작하지 않는다. 이러한 문자렬을 조종하려면 매개 문자렬들에 특수한 구분문자를 써야 하고 그것들을 읽어들일 때

발취연산자가 아니라 getline()함수를 사용해야 한다. 실례 12-4는 공백이 들어있는 문자렬을 출력한다.

(실례 12-4) 파일에로의 문자렬 출력

```
#include <fstream>
using namespace std;
int main()
{
   ofstream outFile("Test.txt");
   outFile << "웅변모임에 출현한 작품들은\n";
   outFile << "세계여 보라\n";
   outFile << "어머님의 당부\n";
   outFile << "백두산은 총대고향\n";
   outFile << "공민적의무를 다하라\n";
   outFile << "백두산총대는 대답하리라\n";
   return 0;
}
```

프로그람을 실행할 때 본문행들을 파일에 써넣는다. 매개 행은 행바꾸기문자('\n')로 끝난다. 이것들은 string클라스의 객체가 아니라 char*문자렬이다. 대부분의 스트림연산은 char*문자렬에 대하여 더 간단히 동작한다.

파일로부터 문자렬을 꺼내기 위하여 ifstream객체를 창조하고 istream의 성원인 getline()함수를 사용하여 한번에 한 행씩 읽을수 있다. getline()함수는 '\n'문자를 만날 때까지 공백을 포함하는 문자들을 읽어들이고 인수로 제공된 완충기에 결과문자렬을 보관한다. 완충기의 최대크기는 둘째 인수로서 주어지며 완충기의 내용은 매개 행에 표시된다.

(실례 12-5) 파일로부터의 문자렬 입력

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
   const int MAX = 80;
   char buffer[MAX];
   ifstream inFile("Test.txt");
   while(!inFile.eof())
   {
      inFile.getline(buffer, MAX);
      cout << buffer << endl;
   }
   return 0;
}
```

실례 12-5의 화면출력은 실례 12-4에 의해 Test.txt파일에 써넣은 자료와 같다. 즉 4개 행의 문자렬이다. 프로그람은 파일에 있는 문자렬의 개수를 미리 알아내는 방

법이 없으므로 파일끝이 나타날 때까지 한 문자렬씩 련속 읽어들인다.

본문파일들을 직접 읽어들이는데 이 프로그람을 사용하지 않는다. 그것은 모든 본문행이 '\n'문자로 완료할것을 요구하고 그렇지 않은 파일을 만나면 프로그람은 오동작하게 된다.

1) 파일끝의 검색

앞에서 본것처럼 ios로부터 파생된 객체들은 조작결과를 검사할수 있는 오유상태기발을 포함한다. 여기서 파일을 조금씩 읽어들일 때 우연히 EOF조건과 만나게 된다. EOF는 읽을 자료가 더는 없을 때 조작체계가 프로그람에 보내는 상수이다. 실례 12-5에서는 다음 행에서 이것을 검사한다.

```
while(!inFile.eof())          // eof를 만날 때까지
```

그러나 eofbit의 검사는 failbit와 badbit와 같은 다른 오유기발들을 엄격하게 검색하지 못한다. 그러므로 스트림조건을 다음과 같이 요구할수 있다.

```
while(inFile.good())   // 오유가 없는 동안
```

또한 스트림을 직접 시험할수 있다. inFile과 같은 스트림객체는 EOF를 비롯한 일반오유조건을 검사할수 있는 값을 가진다. 그러한 조건이 참이면 객체는 값 0을 돌려주고 그렇지 않으면 0아닌 값을 돌려준다. 이 값은 항상 지적자이지만 돌아온 《주소》가 0인가 0아닌 값인가를 시험하는것을 제외하면 의의가 없다. 따라서 while순환을 다음과 같이 쓸수 있다.

```
while(inFile)          // 어떤 오유를 만날 때까지
```

이것은 간단하지만 어떤 오유가 발생하였는가 하는것이 명백하지 않다.

3. 문자입출력

ostream과 istream의 성원들인 put()와 get()함수는 한개문자의 출력과 입력에 쓰인다. 여기에 실례 12-6이 있다. 이 프로그람은 한번에 한 문자씩 문자렬을 출력한다.

(실례 12-6) 파일에로의 문자출력

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string str = "웅변모임에 출현한 작품들은 세계여 보라,"
               "어머님의 당부, 백두산은 총대고향, 공민적의무를 다하라,"
               " 백두산총대는 대답하리라";
   ofstream outFile("Test.txt");
   for(int j=0; j<str.size(); j++)
      outFile.put(str[j]);
   cout << "파일에 출력하였습니다.\n";
   return 0;
}
```

프로그람에서 ofstream객체는 실례 12-4에서처럼 창조된다. string객체 str의 길이는 size()성원함수로 얻으며 for순환의 put()를 사용하여 문자들을 출력한다. 실례 12-7프로그람을 사용하여 파일을 읽어들이고 표시할수 있다.

(실례 12-7) 파일로부터의 문자입력

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
   char ch;
   ifstream inFile("Test.txt");
   while(inFile)
   {
      inFile.get(ch);
      cout << ch;
   }
   cout << endl;
   return 0;
}
```

프로그람은 get()함수에 의하여 EOF에 이를 때까지 계속 읽어들인다. 파일로부터 읽어들인 문자는 cout에 의하여 화면에 표시한다.

파일로부터 문자를 읽어들이는 다른 수법은 ios클라스의 성원인 rdbuf()함수이다. rdbuf()함수는 스트림객체와 련결된 streambuf(또는 filebuf)객체에로의 지적자를 돌려준다. 이 객체는 체계로부터 읽어들인 문자들을 보관하는 완충기를 포함하므로 그것에로의 지적자를 그 자체의 오른쪽에 있는 자료객체처럼 사용할수 있다. 여기에 실례 12-8의 프로그람이 있다.

(실례 12-8) 파일로부터의 문자입력

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
   ifstream inFile("Test.txt");
   cout << inFile.rdbuf();
   cout << endl;
   return 0;
}
```

이 프로그람은 실례 12-7과 같은 효과를 가진다. 또한 가장 짧은 파일지향프로그람으로 된다. 그리고 rdbuf()가 EOF를 만날 때 돌아와야 한다.

4. 2진입출력

형식화된 입출력을 사용하여 여러개의 수값을 디스크에 써넣을수 있으나 대량의

수값자료를 보관하고있으면 문자들의 렬로 수자들을 보관하는것보다 콤퓨터의 RAM기억기에서처럼 보관되는 2진입출력을 사용하는것이 더 효과있다. 2진입출력에서 int는 항상 2byte로 보관되지만 그 본문판에서는 "12345"일 때 5byte를 요구한다. 마찬가지로 float는 항상 4byte로 보관되지만 그 형식화된 판은 "6.02314e13"이므로 10byte를 요구한다.

다음의 실례는 2진형식으로 옹근수배렬을 디스크에 써넣고 기억기에 읽어들이는 방법을 보여준다. 여기서는 두개의 새로운 함수 즉 ofstream의 성원 write()와 ifstream의 성원 read()를 사용한다. 이 함수들은 자료를 바이트 (char형)단위로 취급한다. 여기서는 자료의 형식화방법을 모르며 단순히 디스크파일로부터 완충기에 전송하거나 완충기로부터 디스크파일에 전송한다. write()와 read()의 파라메터는 자료완충기의 주소와 길이이다. reinterpret_cast에 의하여 주소를 char*형으로 강제형변환해야 하며 길이는 완충기의 자료항목수가 아니라 바이트길이(문자수)이다.

(실례 12-9) 옹근수의 2진입출력

```
#include <fstream>
#include <iostream>
using namespace std;
const int MAX = 100;
int buff[MAX];
int main()
{
   for(int j=0; j<MAX; j++) buff[j] = j;
   ofstream os("Edata.dat", ios::binary);
   os.write(reinterpret_cast<char*>(buff), MAX * sizeof(int));
   os.close();
   for(j=0; j<MAX; j++) buff[j] = 0;
   ifstream is("Edata.dat", ios::binary);
   is.read(reinterpret_cast<char*>(buff), MAX * sizeof(int));
   for(j=0; j<MAX; j++)
   {
      if(buff[j] != j)
      {
          cerr << "자료가 정확하지 않습니다.\n";
          return 1;
      }
   }
   cout << "자료는 정확합니다.\n";
   return 0;
}
```

2진자료와 작업할 때에는 write()와 read()의 제2파라메터에서 ios::binary를 사용해야 한다. 이것은 기정의 본문방식이 자료와 함께 일정한 특권을 가지기때문이다. 실례로 본문방식에서 '\n'문자는 2byte(복귀문자와 행바꾸기문자)로 확장되여 디스크에 보관된다. 이것은 TYPE와 같은 DOS기초의 봉사프로그람에 의해 읽어들이기 쉬운 형

식화된 본문으로 만들어지지만 2진파일에 적용할 때 혼란을 일으킨다. 그것은 ASCII 코드값 10을 가지는 매개 바이트가 2byte로 번역되기때문이다. ios::binary인수는 방식비트의 실례이다. open()을 론의할 때 구체적으로 설명한다.

5. reinterpret_cast연산자

실례 12-9에서는 reinterpret_cast연산자에 의하여 read()와 write()함수에서 int형의 완충기가 char형의 완충기처럼 동작하게 한다.

reinterpret_cast연산자는 기억기의 한부분의 형을 변환하므로 신중히 사용해야 한다.

또한 지적자값을 옹근수로 혹은 반대로 변환하는데 사용할수 있다. 이것은 위험한 실천이지만 필요한 경우도 있다.

6. 파일의 닫기

지금까지의 실례들에서는 스트림이 유효범위밖으로 벗어날 때 자동적으로 닫겨지므로 명시적으로 닫을 필요가 없었다. 즉 범위밖으로 벗어날 때 해체자를 호출하고 련결된 파일을 닫았다. 그러나 실례 12-9에서는 출력스트림 os와 입력스트림 is가 같은 파일 Edata.dat와 련결되므로 첫째 스트림은 둘째 스트림을 열기전에 닫는다. 우리는 이미 close()성원함수를 사용하였다.

스트림의 해체자에 관계없이 파일을 닫을 때마다 close()를 명시적으로 사용할수 있다. 이것은 프로그람을 믿음직하고 읽기 쉽게 만든다.

7. 객체의 입출력

C++는 객체지향언어이므로 디스크에서 객체를 입출력하는 과정을 고찰하는것이 중요하다. 다음 실례는 그 과정을 보여준다. 앞에서 사용한 Person클라스가 그 객체를 제공한다.

1) 디스크에 객체의 써넣기

일반적으로 객체를 써넣을 때 2진방식을 사용한다. 이것은 객체를 기억기에 보관하는것과 같은 비트구성으로 디스크에 보관하고 객체안에 포함된 수값자료를 적당히 조종할수 있도록 담보한다. 실례 12-10의 프로그람은 클라스 Person객체에 대한 정보를 사용자로부터 얻어서 디스크파일 person.dat에 써넣는다.

(실례 12-10) 디스크에 Person객체의 보관

```
#include <fstream>
#include <iostream>
using namespace std;
class Person
{
protected:
```

```
    char name[40];
    short age;
public:
    void GetData()
    {
        cout << "\n이름?:";
        cin >> name;
        cout << "\n나이?:";
        cin >> age;
    }
};
int main()
{
    Person pers;
    pers.GetData();
    ofstream outFile("Person.dat", ios::binary);
    outFile.write(reinterpret_cast<char*>(&pers), sizeof(pers));
    return 0;
}
```

Person의 GetData()성원함수는 사용자로부터 정보를 받아들여 pers객체에 보관한다. 아래에 프로그람과의 대화가 있다.

```
이름?:김인철
나이?:23
```

그다음 pers객체의 내용을 write()함수로 디스크에 써넣는다. 이때 sizeof연산자를 사용하여 pers객체의 길이를 얻는다.

2) 디스크로부터 객체의 읽어들이기

read()성원함수를 사용하여 person.dat파일로부터 객체를 읽어들인다.

(실례 12-11) 디스크로부터 Person객체의 읽어들이기

```
#include <fstream>
#include <iostream>
using namespace std;
class Person
{
protected:
    char name[40];
    short age;
public:
    void ShowData()
    {
        cout << "이름=" << name << endl;
        cout << "나이:" << age << endl;
    }
};
int main()
{
```

```
    Person pers;
    ifstream inFile("Person.dat", ios::binary);
    inFile.read(reinterpret_cast<char*>(&pers), sizeof(pers));
    pers.ShowData();
    return 0;
}
```

실례 12-11의 출력은 person.dat파일에 실례 12-10의 프로그람이 어떤 자료를 보관하였는가를 보여준다.

```
이름:김인철
나이:23
```

3) 량립될수 없는 자료구조

파일에 객체를 입출력하는 조작은 같은 클라스의 객체들에게 맡겨야 한다. 실례들에서 클라스 Person의 객체들은 정확히 42byte길이를 가지고 사람의 이름을 표시하는 문자렬에 40byte, 사람의 나이를 표시하는데 short옹근수의 2byte를 각각 할당한다. 만일 두 실례에서 서로 다른 길이를 가진다면 파일을 정확히 읽을수 없다.

그러나 실례 12-10과 실례 12-11의 Person이 같은 자료를 가지지만 다른 성원함수를 가질수 있다.

첫 실례에서는 함수 GetData(), 둘째 실례에서는 ShowData()를 가진다. 성원함수들이 객체나 자료에 따라서 디스크에 써넣지 않으므로 어느 성원함수를 사용하는가 하는데서 문제는 없다. 자료는 같은 형식을 가져야 하지만 성원함수의 불일치는 아무런 효과도 없다. 그러나 이것은 가상함수를 사용하지 않는 단순클라스에서만 참이다.

파생클라스의 객체들을 파일에 써넣고 읽어들이는 경우에는 주의해야 한다. 파생클라스의 객체들은 기억기에서 객체의 자료안에 보관되는 리해할수 없는 수값을 포함한다. 이 수값은 가상함수를 사용할 때 객체의 클라스를 식별하도록 방조한다. 디스크에 객체를 써넣을 때 이 수값은 객체의 다른 자료와 함께 보관된다. 만일 클라스의 성원함수를 변경한다면 이 수값도 물론 변경된다. 파일에 어떤 클라스의 객체를 써넣고 그다음 자료는 같지만 성원함수가 다른 클라스의 객체에로 그것을 읽어들이는 경우에 객체에 대하여 가상함수를 사용하려고 하면 큰 난관에 부닥친다. 대책은 객체에 읽어들이는 클라스가 거기에 써넣는 클라스와 같은가를 확인하는것이다.

8. 여러개의 객체를 사용한 입출력

실례 12-10과 실례 12-11은 한개의 객체만 한번에 써넣고 읽어들인다. 다음의 실례는 파일을 열고 사용자의 요구대로 객체들을 여러개 써넣는다. 그다음 그것들을 읽어들여 파일의 내용을 모두 표시한다.

(실례 12-12) 디스크에로 여러개의 객체들을 써넣고 읽어들이기

```
#include <fstream>
#include <iostream>
using namespace std;
```

```
class Person
{
protected:
   char name[80];
   short age;
public:
   void GetData()
   {
      cout << "\n이름을 입력하십시오:";
      cin >> name;
      cout << "\n나이를 입력하십시오:";
      cin >> age;
   }
   void ShowData()
   {
      cout << "이름=" << name << endl;
      cout << "나이:" << age << endl;
   }
};
int main()
{
   char ch;
   Person pers;
   fstream file;
   file.open("Group.dat", ios::app | ios::out | ios::in | ios::binary);
   do
   {
      cout << "\n개인자료를 입력하시오:";
      pers.GetData();
      file.write(reinterpret_cast<char*>(&pers), sizeof(pers));
      cout << "계속하겠습니까(y/n)? ";
      cin >> ch;
   } while(ch == 'y');
   file.seekg(0);
   file.read(reinterpret_cast<char*>(&pers), sizeof(pers));
   while(!file.eof())
   {
      cout << "\n개인자료:";
      pers.ShowData();
      file.read(reinterpret_cast<char*>(&pers), sizeof(pers));
   }
   cout << endl;
   return 0;
}
```

아래에 출력이 있다.

```
개인자료를 입력하시오:
이름을 입력하십시오:김인철
나이를 입력하십시오:22
```

계속하겠습니까(y/n)? n
개인자료:
이름=리준호
나이=20
개인자료:
이름=최철룡
나이=21
개인자료:
이름=김인철
나이=22

여기서는 파일에 객체가 추가되고 세개 객체의 전체내용이 표시된다.

1) fstream클라스

지금까지 이 장에서 창조한 파일객체는 입력과 출력용이다. 실례 12-12에서는 입출력용 파일을 창조한다. 여기서는 istream과 ostream으로부터 파생되는 iostream으로부터 파생되는 fstream클라스의 객체를 요구한다. 그래야 입력과 출력을 모두 조종할수 있다.

2) open()함수

이 절의 실례에서는 다음 명령문에 의하여 파일객체를 창조하고 초기화한다.

```
ofstream outFile("Test.txt");
```

실례 12-12에서는 다른 수법을 사용한다. 즉 한개 명령문에서 파일을 창조하고 다른 명령문에서 그것을 fstream클라스의 성원인 open()함수로 연다. 이것은 파일열기에서 실패할수 있는 경우에 효과적인 수법이다. 일단 스트림객체를 창조한 다음 새 스트림객체를 창조하지 않고 그것을 다시 연다.

3) 방식비트

앞에서 방식비트 ios::binary를 보았다. open()함수는 여러개의 새로운 방식비트를 포함한다. ios에서 정의된 방식비트는 스트림객체를 여는 각종 방식을 지정한다. 표 12-10은 그 가능성을 보여준다.

표 12-10. open()함수용방식비트

방식비트	결과
in	읽기용으로 연다.(ifstream의 기정값)
out	써넣기용으로 연다.(ofstream의 기정값)
ate	파일끝에서 읽기 시작한다.(AT End)
app	파일끝에서 쓰기 시작한다.(Append)
trunc	파일이 존재한다면 그 길이를 0으로 자른다.(TRUNCate)
nocreate	파일이 이미 존재하지 않으면 열 때 오유.
noreplace	출력용으로 열 때 파일이 이미 존재하면 오유. ate 또는 app가 설정되지 않으면 오유.

방식비트	결과
binary	2진방식으로 파일을 연다.

실례 12-12에서는 이전에 파일에 있던 자료를 보존하는데 ios::app를 사용한다. 즉 파일에 써넣고 프로그람을 완료하였다가 프로그람을 다시 기동하고 파일에 써넣는 경우 현존내용에 추가한다. 파일에 대하여 입출력하는 경우에 in과 out를 사용하고 2진객체를 써넣으려고 하는 경우에 binary를 사용한다. 기발들사이의 내리선(비트별 론리합 |)은 이 기발들을 표시하는 비트가 론리적으로 하나의 옹근수에 결합되게 한다. 따라서 여러개의 비트를 동시에 적용할수 있다.

write()함수에 의하여 파일에 하나의 Person객체를 한번 써넣는다. 써넣기를 끝냈을 때 전체 파일을 읽으려면 seekg()함수에 의하여 파일의 현재위치를 재설정해야 한다. seekg()는 파일의 현재위치를 파일의 선두로 설정한다. 그다음 while순환에서 파일로부터 Person객체를 반복하여 읽어들이고 화면에 표시한다.

이것은 Person객체를 다 읽었다는것을 eof()함수가 발견하고 ios::eofbit의 상태를 돌려줄 때까지 계속한다.

제 4 절. 파일지적자

매개 파일객체는 그와 관련된 두개의 옹근수 즉 얻기지적자와 넣기지적자를 가지고있다. 또한 이것들을 현재얻기위치(current get position)와 현재넣기위치(current put position)라고도 한다. 또는 간단히 현재위치라고도 한다. 현재위치값들은 현재 읽기와 써넣기를 하고있는 장소인 파일안에서의 바이트수를 지적한다.(여기서 지적자는 표준 C++지적자와 다르다.)

보통 현존파일의 선두로부터 마지막 끝까지 읽으려고 한다. 써넣기할 때 파일의 선두로부터 시작하여 현재내용을 지우면서 써넣거나 또는 ios::app방식의 경우에는 파일의 끝에서 써넣기 시작한다. 이것은 기정동작으로서 파일지적자의 조작은 필요없다.

그러나 파일지적자를 자체로 조종하여 파일안의 임의의 위치로부터 읽어들이고 써넣어야 하는 경우가 있다. seekg()와 tellg()함수는 얻기지적자를 설정하고 얻으며 seekp()와 tellp()함수는 넣기지적자에 대하여 같은 조작을 한다.

1. 위치지정

실례 12-12에서 얻기지적자의 실례를 보았다. 여기서 seekg()함수는 얻기지적자를 파일선두로 설정하여 거기로부터 읽기 시작하게 한다. 이 형식의 seekg()는 1인수를 가지고 인수는 파일안에서 절대위치를 표시한다. 파일의 시작은 바이트 0이다. 그림 12-4는 이것을 보여준다.

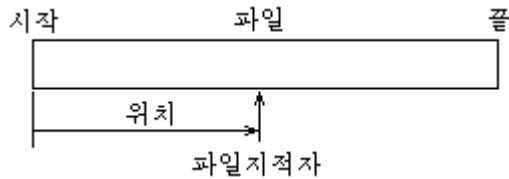



그림 12-4. 1인수를 가지는 seekg()함수

2. 범위지정

seekg()함수는 두가지 방법으로 사용한다. 앞에서 유일한 인수가 파일의 선두로부터의 위치를 표시하는 첫째 방법을 보았다. 또한 두개의 인수를 사용할수도 있다. 여기서 첫 인수는 파일안의 특정위치로부터의 변위를 나타내고 둘째 인수는 변위를 측정하는 위치를 지정한다. 둘째 인수의 가능성은 세가지가 있다. beg는 파일의 시작, cur는 현재 지적자위치, end는 파일의 끝이다. 명령문

```
seekp(-10, ios::end);
```

는 넣기지적자를 파일끝 바로 앞의 10byte로 설정한다. 그림 12-5는 이것을 보여준다.

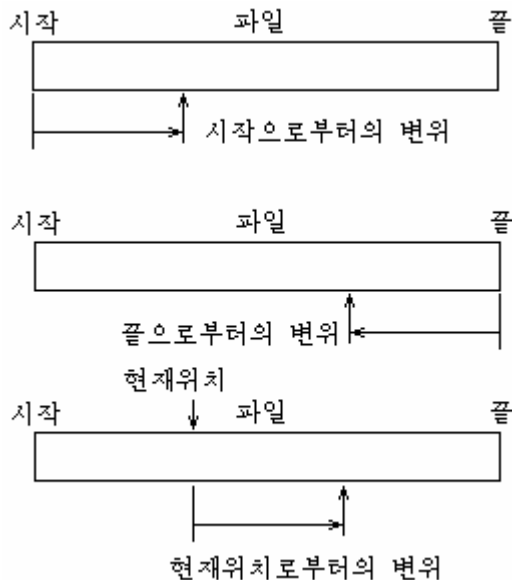



그림 12-5. 2인수를 가지는 seekg()함수

여기에 seekg()의 2인수판을 사용하여 Group.dat파일의 특정한 Person객체를 얻어서 자료를 표시하는 실례가 있다.

(실례 12-13) seekg에 의하여 파일안의 특정한 객체를 읽어들이기

```
#include <fstream>
#include <iostream>
using namespace std;
class Person
{
protected:
```

```
    char name[80];
    short age;
  public:
    void GetData()
    {
      cout << "\n이름을 입력하십시오:"; cin >> name;
      cout << "\n나이를 입력하십시오:"; cin >> age;
    }
    void ShowData()
    {
      cout << "이름=" << name << endl;
      cout << "나이:" << age << endl;
    }
  };
  int main()
  {
    Person pers;
    ifstream inFile;
    inFile.open("Group.dat", ios::in | ios::binary);
    inFile.seekg(0, ios::end);
    int endPosition = inFile.tellg();
    int n = endPosition / sizeof(Person);
    cout << "\n파일에는 " << n << "명의 개인자료가 있습니다.";
    cout << "\n개인의 번호를 입력하시오: "; cin >> n;
    int position = (n - 1) * sizeof(Person);
    inFile.seekg(position);
    inFile.read(reinterpret_cast<char*>(&pers), sizeof(pers));
    pers.ShowData();
    cout << endl;
    return 0;
  }
```

Group.dat파일이 실례 12-12에서 호출한것과 같다고 가정할 때 프로그람의 출력은 다음과 같다.

```
파일에는 세명의 개인자료가 있습니다.
개인의 번호를 입력하시오: 2
이름=김인철
나이=22
```

사용자를 위하여 프로그람은 항목들의 첨수번호를 0으로부터 시작하지만 실제로 1부터 출력하였으므로 Person2는 파일안의 셋중에서 둘째 Person이다.

3. tellg()함수

프로그람이 처음 수행하는 조작은 파일안에 몇명의 Person이 있는가 하는것이다. 명령문 inFile.seekg(0, ios::end);를 사용하여 얻기지적자를 파일의 끝으로 설정한다.

tellg()함수는 얻기지적자의 현재위치를 돌려준다. 프로그람은 tellg()함수에 의하여 현재위치를 얻으며 이것이 바로 바이트수로 된 파일의 길이이다. 다음에 프로그람은

그것을 Person의 크기로 나누어 파일안에 몇명의 Person객체가 있는가를 계산하고 결과를 표시한다.

우의 출력에서 사용자는 파일안의 둘째 객체를 지정하고 프로그람은 seekg()로 이것이 파일안에서 몇바이트위치에 있는가를 계산한 다음 그 점으로부터 시작하여 하나의 Person자료를 읽어들인다. 끝으로 ShowData()로 그 자료를 표시한다.

제 5 절. 파일입출력에서 오유조종

지금까지 파일관련의 실례들에서는 오유가 발생하는 경우를 고찰하지 않았다. 특히 우리는 이미 존재하는 파일들을 열고 써넣기 위해 파일들을 창조하거나 추가한다는것을 가정하였다. 또한 읽고 쓰기할 때 실패가 없는것으로 가정하였다. 실제의 프로그람에서는 그러한 가설들을 확증하고 그것이 부정확하다면 적당한 동작을 취하는것이 중요하다. 자기가 생각하는 파일이 존재하지 않을수도 있고 새로운 파일로 사용하려는 파일이름이 이미 현존파일에 적용되고있을수도 있다. 또한 디스크에 자리가 없을수도 있고 구동기안에 디스크가 없을수도 있다.

1. 오유에 대한 반응

다음의 프로그람은 이러한 오유를 관례적으로 조종하는 방법을 보여준다. 모든 디스크조작은 처리후에 검사된다. 오유가 발생하면 통보문을 출력하고 프로그람은 완료한다. 오유상태를 결정하기 위해 객체자체로부터 돌림값을 검사하는 방법을 처음에 사용하였다. 프로그람은 출력스트림객체를 열고 write()를 한번 호출하여 거기에 전체 옹근수배렬을 써넣고 객체를 닫는다. 그다음 입력스트림객체를 열고 read()를 호출하여 옹근수배렬을 읽어들인다.

(실례 12-14) 입출력오유조종

```
#include <fstream>
#include <iostream>
using namespace std;
#include <process.h>
const int MAX = 1000;
int buff[MAX];
int main()
{
  for(int j=0; j<MAX; j++)
     buff[j] = j;
  ofstream os;
  os.open("adata.dat", ios::trunc | ios::binary);
  if(!os)
  {
     cerr << "출력파일을 열수 없습니다.\n";
```

```
        exit(1);
    }
    cout << "써넣기중입니다...\n";
    os.write(reinterpret_cast<char*>(buff), MAX * sizeof(int));
    if(!os)
    {
        cerr << "파일에 써넣을수 없습니다.\n";
        exit(1);
    }
    os.close();
    for(j=0; j<MAX; j++)
        buff[j] = 0;
    ifstream is;
    is.open("adata.dat", ios::binary);
    if(!is)
    {
        cerr << "입력파일을 열수 없습니다.\n";
        exit(1);
    }
    cout << "읽기중입니다...\n";
    is.read(reinterpret_cast<char*>(buff), MAX * sizeof(int));
    if(!is)
    {
        cerr << "파일로부터 읽을수 없습니다.\n";
        exit(1);
    }
    is.close();
    for(j=0; j<MAX; j++)
    {
        if(buff[j] != j)
        {
            cerr << "\n자료가 부정확합니다.\n";
            exit(1);
        }
    }
    cout << "\n자료가 정확합니다.\n";
    return 0;
}
```

2. 오유해석

실례 12-14에서는 전체 스트림객체의 돌림값을 검사하는 방법으로 입출력조작에서 오유가 발생하였는가를 결정한다.

```
if(is)
    // 오유가 발생함
```

여기서 is는 제대로 실행되면 지적자를 돌려주고 제대로 실행되지 안되면 0을 돌려준다. 즉 오유가 무엇이든 같은 방법으로 오유를 검색하고 같은 동작을 취한다. 그

러나 ios오유상태기발을 사용하여 파일입출력오유에 대한 구체적인 정보를 얻을수도 있다. 이미 화면과 건반입출력에서 동작하는 상태기발들을 보았다. 실례 12-15는 상태기발을 파일입출력에서 사용하는 방법을 보여준다.

(실례 12-15) 파일열기오유검사

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
  ifstream file;
  file.open("atest.dat");
  if(!file)
     cout << "\ngroup.dat파일을 열수 없습니다.";
  else
     cout << "\n파일을 열었습니다.";
  cout << "\n파일=" << file;
  cout << "\n오유상태=" << file.rdstate();
  cout << "\ngood()=" << file.good();
  cout << "\neof()=" << file.eof();
  cout << "\nfail()=" << file.fail();
  cout << "\nbad()=" << file.bad();
  file.close();
  cout << endl;
  return 0;
}
```

이 프로그람은 우선 목적파일의 값을 검사한다. 그 값이 0이면 파일은 존재하지 않으므로 열리지 않는다. 여기에 실례 12-15의 출력이 있다.

```
group.dat파일을 열수 없습니다.";
파일=0x1c730000
오유상태=4
good()=0
eof()=0
fail()=4
bad()=4
```

rdstate()에 의해 돌아온 오유상태는 4이다. 이것은 파일이 존재하지 않는다는것을 가리키는 비트로서 1로 설정된다. 다른 비트들은 모두 0으로 설정된다. good()함수는 어떤 비트도 설정되지 않았을 때에 1(true)을 돌려주고 그렇지 않으면 0(false)를 돌려준다. EOF가 아니므로 eof()는 0을 돌려준다. fail()과 bad()함수는 오유가 발생하면 0 아닌 값을 돌려준다.

일련의 프로그람들에서는 매번의 입출력조작후에 일부 함수들을 리용하여 기대한 대로 실행되였다는것을 담보하여야 한다.

제 6 절. 성원함수에 의한 파일입출력

지금까지는 main()함수에서 파일입출력의 상태를 조종하였다. 복잡한 클라스들을 사용할 때에는 파일입출력조작을 클라스의 성원함수로 포함하는것이 자연스럽다.

이 절에서는 이것을 수행하는 두개의 프로그람을 고찰한다.

우선 매개 객체가 파일에 대한 읽기와 쓰기에 응답할수 있는 일반성원함수를 사용한다.

다음으로 정적성원함수가 클라스의 모든 객체를 한번에 어떤 방법으로 읽고 쓸수 있는가를 보여준다.

1. 자기자체를 읽고 쓰는 객체

때때로 클라스의 매개 객체가 파일에로 그 자체를 써넣고 읽어들일수 있다. 이것은 간단한 방법으로서 한번에 써넣거나 읽어들이면 객체가 여러개가 아닐 때 잘 동작한다. 이 실례에서는 Person클라스에 성원함수 DiskOut()와 DiskIn()을 추가한다. 이 함수들은 Person객체자체가 디스크에 써넣고 그 자체를 읽어들이게 한다.

몇가지 간단한 가설을 만들자.

첫째로, 클라스의 모든 객체는 같은 파일 persFile.dat에 보관된다.

둘째로, 새로운 객체는 항상 파일의 끝에 추가된다.

DiskIn()함수에 대한 인수는 그 파일안에서 임의의 Person에 대한 자료를 읽어들이게 한다. 파일끝에 있는 자료를 읽으려는 시도를 방지하기 위하여 파일에 보관된 Person의 수를 돌려주는 정적성원함수 DiskCount()를 포함한다. 여기에 실례 12-16의 프로그람이 있다.

(실례 12-16) Person객체에 의한 디스크입출력

```
#include <fstream>
#include <iostream>
using namespace std;
class Person
{
protected:
   char name[40];
   short age;
public:
   void GetData()
   {
      cout << "\n  이름을 입력하십시오:"; cin >> name;
      cout << "\n  나이를 입력하십시오:"; cin >> age;
   }
   void ShowData()
   {
```

```
        cout << "  이름=" << name << endl;
        cout << "  나이=" << age << endl;
    }
    void DiskIn(int);
    void DiskOut();
    static int DiskCount();
};
void Person::DiskIn(int pn)
{
    ifstream inFile;
    inFile.open("PersFile.dat", ios::binary);
    inFile.seekg(pn * sizeof(Person));
    inFile.read((char*)this, sizeof(*this));
}
void Person::DiskOut()
{
    ofstream outFile;
    outFile.open("PersFile.dat", ios::app | ios:: binary);
    outFile.write((char*)this, sizeof(*this));
}
int Person::DiskCount()
{
    ifstream inFile;
    inFile.open("PersFile.dat", ios::binary);
    inFile.seekg(0, ios::end);
    return (int)inFile.tellg() / sizeof(Person);
}
int main()
{
    Person p;
    char ch;
    do
    {
        cout << "개인자료를 입력하시오:"; p.GetData();
        p.DiskOut();
        cout << "계속하겠습니까(y/n)?: "; cin >> ch;
    } while(ch == 'y');
    int n = Person::DiskCount();
    cout << "파일에는 " << n << "명의 개인자료가 있습니다.\n";
    for(int j=0; j<n; j++)
    {
        cout << endl << j <<"번째 개인자료 ";
        p.DiskIn(j);
        p.ShowData();
    }
    cout << endl;
    return 0;
}
```

여기에는 복잡한것이 없다. 이 프로그람의 대부분의 요소를 이미 보았다. 프로그람은 실례 12-12와 같은 방법으로 동작한다. 그러나 디스크조작의 세부는 main()에서 볼수 없고 Person클라스에 은페되여있다.

써넣거나 읽어들이려는 자료가 어디에 있는지 미리 알수 없으므로 매개 객체는 기억기안의 서로 다른 위치에 놓여있다. 그러나 성원함수안에서 항상 this지적자가 자기의 주소를 알려준다. read()와 write()스트림함수들에서 읽어들이거나 써넣으려는 객체의 주소는 this이고 그 크기는 sizeof(*this)이다.

프로그람을 실행할 때 이미 두명의 Person이 파일안에 있다고 가정하면 출력은 다음과 같다.

```
  이름을 입력하십시오:김인수
  나이를 입력하십시오:20
계속하겠습니까(y/n)?:y
  이름을 입력하십시오:리광호
  나이를 입력하십시오:21
계속하겠습니까(y/n)?:n
41번째 개인자료 ";
  이름=최철수
  나이=24
42번째 개인자료 ";
  이름=김은희
  나이=19
43번째 개인자료 ";
  이름=김인수
  나이=20
44번째 개인자료 ";
  이름=리광호
  나이=21
```

사용자가 클라스에서 쓰이는 파일이름을 지정할수 있게 하려면 정적성원변수(말하자면 char filename[])를 창조하고 정적함수가 그것을 설정하게 해야 한다. 혹은 매개 객체와 련관된 파일의 이름을 비정적함수를 사용하여 만들수 있다.

2. 자기자체를 읽고 쓰는 클라스

기억기에 객체가 많고 그것들을 모두 파일에 써넣으려고 한다. 실례 12-16과 같이 매개 객체마다 파일을 열고 거기에 한개 객체를 써넣고 그것을 닫을 때마다 성원함수를 호출하는것은 비효과적이다. 파일을 한번 열고 거기에 객체를 모두 써넣고 그것을 닫는것이 훨씬 더 빠르다.

1) 정적함수

많은 객체를 한번에 쓰는 한가지 방도는 정적성원함수를 사용하는것이다. 정적성원함수는 매개 객체별로가 아니라 클라스전체에 적용된다. 정적성원함수는 모든 객체들을 한번에 써넣을수 있다. 그러면 모든 객체들이 어디에 있는가를 알아내는 그러한

함수는 어떤것인가?

그 함수는 정적자료로 보관할수 있는 객체들에로의 지적자배렬을 호출할수 있다. 매개 객체가 창조되면 그것에로의 지적자가 이 배렬에 보관된다. 또한 정적자료성원에 창조한 객체의 수를 보관한다. 정적써넣기함수는 파일을 열고 배렬을 순환하면서 매개 객체를 차례로 써넣고 마지막에 파일을 닫는다.

2) 파생된 객체들의 크기

기억기에 보관된 객체들이 서로 다른 크기를 가진다고 하자.

이런 경우에는 대체로 여러개의 클라스들이 기초클라스로부터 파생될 때 제기된다. 실례로 실례 9-5를 고찰하자.

여기에는 Manager, Scientist, Labour클라스들의 기초클라스로서 동작하는 Employee클라스가 있다. 세개의 파생클라스의 객체들은 각이한 크기를 가지므로 서로 다른 량의 자료를 포함한다. 특히 이름과 종업원번호와 같이 모든 종업원에 적용되는 자료들외에 Manager에게는 title이 있고 Scientist에게는 pubs가 있다.

간단한 순환과 ofstream의 write()성원함수를 사용하여 세가지 파생형(Manager, Scientist, Labour)의 객체를 포함하는 목록으로부터 자료를 써넣고 싶을수 있다. 그러나 이 함수를 사용하려면 객체의 크기를 알아야 하므로 그것을 둘째 인수로 한다.

Employee형객체에로의 지적자배렬(arrap[])이 있다고 하자.

이 지적자들은 세개의 파생클라스의 객체들을 지적할수 있다.(실례 11-4의 프로그람에서 파생클라스의 객체에로의 지적자배렬을 주었다.)

가상함수를 사용하면 다음과 같은 명령문을 쓸수 있다.

```
arrap[j]->PutData();
```

지적자가 가리키는 객체에 부합되는 PutData()함수판이 기초클라스의 함수대신 사용된다.

또한 지적자인수의 크기를 얻는데 sizeof()함수를 사용할수 있는가?

즉

```
out.write( (char*)arrap[j], sizeof(*arrap[j])); // 좋지 않다.
```

를 사용할수 없다. sizeof()는 가상함수가 아니고 지적자형보다도 지적하는 객체의 형을 고려해야 한다. sizeof()는 항상 기초클라스객체의 크기를 돌려준다.

3) typeid()함수의 사용

우리가 가지고있는것이 모두 객체에로의 지적자라면 그 크기를 어떻게 얻겠는가?

이에 대한 한가지 대답은 2장에서 소개한 typeid()함수이다. typeid()를 사용하려면 번역프로그람의 실행시형정보선택을 가능하게 해야 한다.

다음 실례는 그 동작을 보여준다. 일단 객체의 크기를 알면 write()함수에서 그것을 사용하여 디스크에 객체를 써넣을수 있다.

실례 9-5프로그람에 간단한 대면부를 추가하고 고유한 성원함수를 가상함수로 하

여 객체에로의 지적자배렬을 사용할수 있다.

또한 마지막 절에서 설명한 오유검사기술의 일부를 사용한다.

이것은 좀 모호한 프로그람이지만 완전규모의 자료기지응용프로그람에서 사용할수 있는 많은 기술을 보여준다.

또한 객체지향프로그람작성법의 실제능력을 보여준다.

그러면 어떻게 하면 한 파일에 다른 크기의 객체를 써넣는데 단일명령문을 사용할수 있겠는가?

여기에 실례 12-7이 있다.

(실례 12-17) 종업원객체에 대한 파일입출력

```
#include <fstream>
#include <iostream>
#include <typeinfo>
using namespace std;
#include <process.h>
const int LEN = 30;
const int MAXEMLEN = 100;
enum EmployeeType { TMANAGER, TSCIENTIST, TLABORER };
class Employee
{
private:
   char name[LEN];
   unsigned long number;
   static int n;
   static Employee* arrap[];
public:
   virtual void GetData()
   {
      cin.ignore(10, '\n');
      cout << "\n  이름? "; cin >> name;
      cout << "\n  종업원 번호? "; cin >> number;
   }
   virtual void PutData()
   {
      cout << "\n  이름: " << name;
      cout << "\n  종업원 번호: " << number;
   }
   virtual EmployeeType GetType();
   static void Add();
   static void Display();
   static void Read();
   static void write();
};
int Employee::n;
Employee* Employee::arrap[MAXEMLEN];
class Manager : public Employee
```

```
{
private:
   char title[LEN];
public:
   void GetData()
   {
      Employee::GetData();
      cout << "   직위? "; cin >> title;
   }
   void PutData()
   {
      Employee::PutData();
      cout << "\n   직위: " << title;
   }
};
class Scientist : public Employee
{
private:
   int pubs;
public:
   void GetData()
   {
      Employee::GetData();
      cout << "   출판물 건수? "; cin >> pubs;
   }
   void PutData()
   {
      Employee::PutData();
      cout << "\n   출판물 건수: " << pubs;
   }
};
class Laborer : public Employee { };
void Employee::Add()
{
   char ch;
   cout << "관리일군을 추가하려면 'm'을 누르시오"
           "\n기사를 추가하려면 's'를 누르시오"
           "\n로동자를 추가하려면 'l'을 누르시오"
           "\n이상의것들중에서 하나 선택하시오:";
   cin >> ch;
   switch(ch)
   {
   case 'm': arrap[n] = new Manager; break;
   case 's': arrap[n] = new Scientist; break;
   case 'l': arrap[n] = new Laborer; break;
   default:
      cout << "\n알려지지 않은 종업원형입니다.\n"; return;
   }
```

```
   arrap[n++]->GetData();
}
void Employee::Display()
{
   for(int j=0; j<n; j++)
   {
      cout << (j + 1);
      switch(arrap[j]->GetType())
      {
      case TMANAGER: cout << "형은 관리일군"; break;
      case TSCIENTIST: cout << "형은 기사"; break;
      case TLABORER: cout << "형은 로동자"; break;
      default:
          cout << "알려지지 않은 형";
      }
      arrap[j]->PutData();
      cout << endl;
   }
}
EmployeeType Employee::GetType()
{
   if(typeid(*this) == typeid(Manager))
      return TMANAGER;
   else if(typeid(*this) == typeid(Scientist))
      return TSCIENTIST;
   else if(typeid(*this) == typeid(Laborer))
      return TLABORER;
   else
   {
      cerr << "\n종업원형이 옳지 않습니다.";
      exit(1);
   }
   return TMANAGER;
}
void Employee::write()
{
   int size;
   cout << n << "건의 종업원자료를 써넣고 있습니다.\n";
   ofstream ouf;
   EmployeeType eType;
   ouf.open("EMPLOY.DAT", ios::trunc | ios::binary);
   if(!ouf)
   {
      cout << "\n파일을 열수 없습니다.\n";
      return;
   }
   for(int j=0; j<n; j++)
   {
```

```
        eType = arrap[j]->GetType();
        ouf.write((char*)&eType, sizeof(eType));
        switch(eType)
        {
        case TMANAGER: size = sizeof(Manager); break;
        case TSCIENTIST: size = sizeof(Scientist); break;
        case TLABORER: size = sizeof(Laborer); break;
        }
        ouf.write((char*)(arrap[j]), size);
        if(!ouf)
        {
            cout << "\n파일에 써넣을수 없습니다.\n";
            return;
        }
    }
}
void Employee::Read()
{
    int size;
    EmployeeType eType;
    ifstream inf;
    inf.open("EMPLOY.DAT", ios::binary);
    if(!inf)
    {
        cout << "\n파일을 열수 없습니다.\n";
        return;
    }
    n = 0;
    while(true)
    {
        inf.read((char*)&eType, sizeof(eType));
        if(inf.eof())
            break;
        if(!inf)
        {
            cout << "\n파일로부터 형을 읽을수 없습니다.\n";
            return;
        }
        switch(eType)
        {
        case TMANAGER:
            arrap[n] = new Manager;
            size = sizeof(Manager);
            break;
        case TSCIENTIST:
            arrap[n] = new Scientist;
            size = sizeof(Scientist);
            break;
```

```
        case TLABORER:
            arrap[n] = new Laborer;
            size = sizeof(Laborer);
            break;
        default:
            cout << "\n파일에 알려지지 않은 형이 있습니다.\n";
            return;
        }
        inf.read((char*)arrap[n], size);
        if(!inf)
        {
            cout << "\n파일로부터 자료를 읽을수 없습니다.\n";
            return;
        }
        n++;
    }
    cout << n << "건의 종업원자료를 읽는중입니다.\n";
}
int main()
{
    char ch;
    while(true)
    {
        cout << "'a'--종업원 1명을 추가한다"
             << "\n'd'--모든 종업원자료를 표시한다"
             << "\n'w'--모든 종업원자료를 파일에 써넣는다"
             << "\n'r'--모든 종업원자료를 파일로부터 읽어들인다"
             << "\n'x'--완료"
             << "\n항목을 선택하시오: ";
        cin >> ch;
        switch(ch) {
        case 'a':
            Employee::Add(); break;
        case 'd':
            Employee::Display(); break;
        case 'w':
            Employee::write(); break;
        case 'r':
            Employee::Read(); break;
        case 'x':
            exit(0);
        default:
            cout << "\n해석할수 없는 지령입니다.\n";
        }
    }
    return 0;
}
```

4) 객체형에 대한 코드번호

이제는 기억기안에서 객체의 클라스를 찾는 방법은 알고있다. 그러면 디스크로부터 읽으려고 하는 객체의 클라스를 어떻게 알아내겠는가?

이것을 방조하는 함수는 없다. 디스크에 객체의 자료를 써넣을 때 디스크에 객체의 자료앞에 직접 코드번호(enum변수 EmployeeType)를 써넣어야 한다. 그다음 파일로부터 기억기에로 객체를 읽어들일 때 그 값을 읽어들이고 지적된 형의 새로운 객체를 창조한다. 끝으로 파일로부터 새로운 객체에 자료를 복사한다.

5) 비법적인 객체를 창조할수 없다

우연히 객체의 자료를 어떤 장소, 말하자면 char형배렬에로 읽어들인 다음 객체에로의 지적자가 그 령역을 가리키도록 설정할수 있다. 강제형변환을 사용하여 그것을 만들자면

```
char someArray[MAX];
AClass* aPtrToObj;
aPtrToObj = reinterpret_cast<AClass*>(someArray)); // 이렇게 하지 않는다.
```

그러나 이것은 객체를 창조하지 않고 지적자가 어떤 객체를 가리키는것처럼 사용하려고 한다. 객체창조에는 오직 두가지 합법적인 방법이 있다.

우선 번역시에 객체를 명시적으로 정의하는것이다.

```
AClass anObj;
```

또한 실행시에 new에 의하여 객체를 창조하고 지적자에 그 위치를 대입하는것이다.

```
aPtrToObj = new AClass;
```

객체를 창조할 때 구성자가 호출된다. 이것은 구성자를 정의하지 않았을 때에도 요구되고 이때에는 기정구성자가 쓰인다. 객체는 그 안에 자료를 가지고있는 기억령역이상이며 또한 성원함수모임(그 일부를 보지 못했지만)이기도 하다.

6) 실례 12-17과의 대화

여기에 프로그람과의 간단한 대화가 있다. 여기서는 Manager, Scientist, Labour를 각각 하나씩 기억기에 창조하고 디스크에 그것들을 써넣고 그다음 반대로 읽어들이고 표시한다. (여기서 겹치는 이름과 부서는 허용되지 않는다.)

```
'a'--종업원 1명을 추가한다
'd'--모든 종업원자료를 표시한다
'w'--모든 종업원자료를 파일에 써넣는다
'r'--모든 종업원자료를 파일로부터 읽어들인다
'x'--완료
항목을 선택하시오: a
관리일군을 추가하려면 'm'을 누르시오
기사를 추가하려면 's'를 누르시오
로동자를 추가하려면 'l'을 누르시오
이상의것들중에서 하나 선택하시오: m
  이름? 김호철
  종업원 번호? 1111
  직위? 지배인
```

'a'--종업원 1명을 추가한다
'd'--모든 종업원자료를 표시한다
'w'--모든 종업원자료를 파일에 써넣는다
'r'--모든 종업원자료를 파일로부터 읽어들인다
'x'--완료
항목을 선택하시오: a
관리일군을 추가하려면 'm'을 누르시오
기사를 추가하려면 's'를 누르시오
로동자를 추가하려면 'l'을 누르시오
이상의것들중에서 하나 선택하시오: s
 이름? 리준호
 종업원 번호? 2222
 출판물 건수? 50
'a'--종업원 1명을 추가한다
'd'--모든 종업원자료를 표시한다
'w'--모든 종업원자료를 파일에 써넣는다
'r'--모든 종업원자료를 파일로부터 읽어들인다
'x'--완료
항목을 선택하시오: a
관리일군을 추가하려면 'm'을 누르시오
기사를 추가하려면 's'를 누르시오
로동자를 추가하려면 'l'을 누르시오
이상의것들중에서 하나 선택하시오: l
 이름? 조연
 종업원 번호? 3333
'a'--종업원 1명을 추가한다
'd'--모든 종업원자료를 표시한다
'w'--모든 종업원자료를 파일에 써넣는다
'r'--모든 종업원자료를 파일로부터 읽어들인다
'x'--완료
항목을 선택하시오: w
3건의 종업원자료를 써넣고있습니다.
'a'--종업원 1명을 추가한다
'd'--모든 종업원자료를 표시한다
'w'--모든 종업원자료를 파일에 써넣는다
'r'--모든 종업원자료를 파일로부터 읽어들인다
'x'--완료
항목을 선택하시오: r
3건의 종업원자료를 읽는중입니다.
'a'--종업원 1명을 추가한다
'd'--모든 종업원자료를 표시한다
'w'--모든 종업원자료를 파일에 써넣는다
'r'--모든 종업원자료를 파일로부터 읽어들인다
'x'--완료
항목을 선택하시오: d
형은 관리일군
이름: 김호철
종업원 번호: 1111

직위: 지배인
형은 기사
이름: 리준호
종업원 번호: 2222
출판물 건수: 50
형은 로동자
이름: 조연
종업원 번호: 3333

물론 디스크에 자료를 써넣은 다음 완료하고 프로그람을 다시 실행하여 반대로 읽어들이고 모든 자료를 표시할수 있다.

파일로부터 하나의 종업원자료를 삭제하고 하나의 종업원자료를 얻고 매개 종업원에 대하여 어떤 특성을 탐색하는것과 같은 함수들을 프로그람에 추가하는것은 간단한다.

제 7 절. 발취 및 삽입연산자의 재정의

여기서 또 하나의 스트림관련항목 즉 발취연산자와 삽입연산자의 재정의를 고찰하자. 이것은 C++의 강력한 특성으로서 int와 double과 같은 기본형과 꼭같이 사용자정의자료형에 대해서도 입출력할수 있게 한다.

실례로 cdl이라는 Crawdad클라스의 객체가 있다면 명령문

```
cout << "\ncd1=" << cd1;
```

에 의해 기본형처럼 표시할수 있다.

발취와 삽입연산자를 재정의하면 영상표시장치와 건반이 독립적으로 작업할수 있다. 그리고 좀 더 구체적으로 재정의하면 디스크파일과 작업할수 있다.

이 두가지 경우의 실례를 고찰하자.

1. cout와 cin을 위한 재정의

실례 12-18은 Distance클라스의 삽입연산자와 발취연산자를 재정의하여 그것들이 cout와 cin과 작업할수 있게 한다.

(실례 12-18) <<와 >>연산자의 재정의

```
#include <iostream>
using namespace std;
class Distance
{
private:
   int meters;
   float centies;
public:
   Distance() : meters(0), centies(0.0) {}
   Distance(int me, float ce) : meters(me), centies(ce) { }
```

```
    friend istream& operator>> (istream& s, Distance& d);
    friend ostream& operator<< (ostream& s, Distance& d);
};
istream& operator >> (istream& s, Distance& d)
{
    cout << "\n메터를 입력하시오:"; s >> d.meters;
    cout << "센치메터를 입력하시오:"; s >> d.centies;
    return s;
}
ostream& operator << (ostream& s, Distance& d)
{
    cout << d.meters << "m " << d.centies << "cm";
}
int main()
{
    Distance dist1, dist2;
    Distance dist3(11, 6.25);
    cout << "\n두개의 거리를 입력하시오:";
    cin >> dist1 >> dist2;
    cout << "\ndist1=" << dist1 << "\ndist2=" << dist2;
    cout << "\ndist3=" << dist3 << endl;
    return 0;
}
```

이 프로그람은 사용자로부터 두개의 Distance값을 얻어서 그 값들과 프로그람에서 초기화한 다른 값들을 출력한다. 프로그람과의 대화는 다음과 같다.

```
두개의 거리를 입력하시오:
메터를 입력하시오:10
센치메터를 입력하시오:3.5
메터를 입력하시오:12
센치메터를 입력하시오:6
메터를 입력하시오:10
센치메터를 입력하시오:3.5
dist1=10m 3.5cm
dist2=12m 6cm
dist3=11m 6.25cm
```

명령문

```
cin >> dist1 >> dist2;
```

과

```
cout << "\ndist1" <<dist1 << "\ndist2" <<dist2;
```

을 사용하면 Distance객체들을 다른 자료형처럼 아주 관례적이고 자연스럽게 취급할 수 있다.

<<와 >>연산자들은 류사한 방법으로 재정의된다. 이 연산자들은 istream(>>일 때) 또는 ostream(<<일 때)의 객체를 참고에 의해 돌려준다. 돌림값은 다중입력과 다중출력을 가능하게 한다. 연산자들은 참고에 의해 넘어온 두개의 인수를 가진다. >>의 첫

인수는 istream의 객체(cin과 같다.)이고 <<인 경우에는 ostream의 객체(cout와 같다.)이다. 둘째 인수는 표시하려는 클라스(이 실례에서는 Distance)의 객체이다. >>연산자는 첫째 인수에 지정된 스트림으로부터 입력을 받아들여 제2인수로서 지적된 객체의 성원자료에 넣는다. <<연산자는 제2인수로서 지정된 객체로부터 자료를 삭제하고 그것을 제1인수로 지적된 스트림에 보낸다.

operator<<()와 operator>>()함수들은 Distance클라스의 동료이여야 한다. 그것은 istream과 ostream객체들이 연산자의 왼변에 나타나기때문이다.

이와 같은 방법으로 다른 클라스들의 삽입연산자와 발취연산자를 재정의할수 있다.

2. 파일을 위한 재정의

다음 실례는 cout와 cin은 물론 파일입출력이 가능하도록 Distance클라스의 <<와 >>연산자를 재정의한다.

(실례 12-19) 파일과 작업하는 <<와 >>연산자의 재정의

```
#include <fstream>
#include <iostream>
using namespace std;
class Distance
{
private:
    int meters;
    float centies;
public:
    Distance() : meters(0), centies(0.0) {}
    Distance(int me, float ce) : meters(me), centies(ce) { }
    friend istream& operator>> (istream& s, Distance& d);
    friend ostream& operator<< (ostream& s, Distance& d);
};
istream& operator >> (istream& s, Distance& d)
{
    char dummy;
    s >> d.meters >> dummy >> d.centies >> dummy >> dummy;
    return s;
}
ostream& operator << (ostream& s, Distance& d)
{
    cout << d.meters << "m " << d.centies << "cm";
}
int main()
{
    char ch;
    Distance dist1;
    ofstream oFile;
    oFile.open("Dist.dat");
```

```
do
{
    cout << "\n거리를 입력하시오:";
    cin >> dist1;
    oFile << dist1;
    cout << "계속하겠습니까(y/n)?";
    cin >> ch;
} while(ch != 'n');
oFile.close();
ifstream iFile;
iFile.open("Dist.dat");
cout << "\n디스크파일의 내용:\n";
while(true)
{
    iFile >> dist1;
    if(iFile.eof())
        break;
    cout << "거리=" << dist1 << endl;
}
return 0;
}
```

앞에서 재정의한 연산자를 약간 변경하였다. >>연산자는 입력에 대한 재촉문을 출력하지 않는다. 왜냐하면 파일에 재촉문을 낼 필요가 없기때문이다. 이때 사용자가 메터와 센치메터값을 서로 구분하여 입력하는 방법을 정확히 아는것으로 가정한다. <<연산자는 변경하지 않는다. 프로그람은 사용자로부터 입력을 얻어서 열려있는 파일에 매개 Distance값을 써넣는다. 사용자가 입력을 끝내면 프로그람은 파일로부터 모든 값들을 읽어들이고 표시한다. 여기에 프로그람의 출력이 있다.

```
거리를 입력하시오:3m 4.5cm
계속하겠습니까(y/n)?y
거리를 입력하시오:7m 11.25cm
계속하겠습니까(y/n)?y
거리를 입력하시오:11m 6cm
계속하겠습니까(y/n)?n
디스크파일의 내용:
거리=3m 4.5cm
거리=7m 11.25cm
거리=11m 6cm
```

거리는 파일에 한 문자씩 보관된다. 이 실례에서 파일의 내용은 다음과 같다.

```
3m 4.5cm7m 11.25cm11m 6cm
```

사용자가 정확히 구분기호를 입력하지 못하여 거리입력에서 실패하면 그것을 파일에 정확히 써넣을수 없으며 <<연산자로 파일을 읽어들일수 없다. 실례프로그람의 오유검사에서는 입력이 중요하다.

제 8 절. 스트림객체로서의 기억기

기억기의 한 부분을 스트림객체로 취급할수 있다. 즉 파일의 자료를 기억기에 삽입할수 있다. 이것은 특별한 방법으로 출력을 형식화할 때 (즉 소수점의 오른쪽에 정확히 두자리만 표시하는 경우 등) 사용할수 있다.

또한 입력할 때 문자렬을 요구하는 본문출력함수를 사용할 필요가 있다.

이것은 일반적으로 Windows와 같은 도형방식사용자대면부환경에서 출력함수를 호출할 때 요구된다. 그것은 출력함수가 자주 문자렬을 인수로서 요구하기때문이다. (C프로그람작성자들은 이 목적에 sprintf()함수를 사용한다.)

스트림클라스는 이러한 기억기안의 형식화를 제공한다. 기억기에로의 출력을 위하여 ostream의 파생클라스 ostrstream가 제공되고 기억기에로의 입력을 위하여 istream의 파생클라스 istrstream가 제공된다. 입출력용의 기억기객체에 대해서는 iostream의 파생클라스 strstream이 제공된다.

대체로 ostrstream을 사용한다. 다음의 실례에서 그 방법을 보여준다. 우선 기억기안에 자료완충기를 만드는것으로부터 시작한다. 그다음 스트림의 구성자에 인수로서 기억완충기와 그 크기를 넘기여 ostrstream객체를 창조한다. 그러면 스트림객체처럼 기억완충기에 형식화된 본문을 출력할수 있다. 여기에 실례 12-20의 프로그람이 있다.

(실례 12-20) 형식화된 자료를 기억기에 써넣기

```
#include <strstream>
#include <iostream>
#include <iomanip>
using namespace std;
const int SIZE = 80;
int main()
{
  char ch = 'x';
  int j = 77;
  double d = 67890.12345;
  char str1[] = "Kafta";
  char str2[] = "Freud";
  char memBuff[SIZE];
  ostrstream oMem(memBuff, SIZE);
  oMem << "ch=" << ch << endl << "j=" << j << endl
         << setiosflags(ios::fixed) << setprecision(2)
         << "d=" << d << endl << "str1=" << str1 << endl
         << "str2=" << str2 << endl << ends;
  cout << memBuff;
  return 0;
}
```

프로그람을 실행할 때 memBuff는 형식화된 본문으로 채워진다.

```
ch=x\nj=77\nd=67890.12\nstr1=Kafta\nstr2=Freud\n\0
```

보통 방법으로 류동소수점수를 형식화할수 있다. ios::fixed를 사용하여 고정소수형식을 지정하고 setprecision()을 사용하여 소수점아래 두자리를 지정한다. 조작자 ends는 문자렬의 끝에 '\0'문자를 삽입하여 EOF를 제공한다. cout에 의하여 보통 방법으로 이 완충기를 표시하면 다음과 같다.

```
ch=x
j=77
d=67890.12
str1=Kafta
str2=Freud
```

실례에서는 완충기를 보기 위해 그 내용을 표시한다.

제 9 절. 지령행인수

MS-DOS를 사용해본적이 있으면 프로그람을 호출할 때 지령행인수와 친숙되였을 것이다. 지령행인수(command line argument)는 응용프로그람에 자료파일의 이름을 넘길 때 많이 사용된다. 실례로 문서처리프로그람을 호출할수 있다.

```
C:>WordProc afile.doc
```

여기서 afile.doc는 지령행인수이다.

그러면 C++프로그람이 지령행인수를 어떻게 읽어들이는가?

실례 12-21은 지령행인수를 보여준다.

(실례 12-21) 지령행인수

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
   cout << "\nargc=" << argc << endl;
   for(int j=0; j<argc; j++)
      cout << j << "번째 인수 = " << argv[j] << endl;
   return 0;
}
```

프로그람의 출력이 있다.

```
C:\PDL\Chapt12\EX1221 uno dos tres
argc=4
0번째 인수= C:\PDL\Chapt12>EX1221
1번째 인수=uno
2번째 인수=dos
3번째 인수=tres
```

지령행인수를 읽어들이려면 main()함수에는 두개의 인수가 있어야 한다. 우선 argc(argument count)는 지령행인수의 총 개수를 표시한다. 첫 지령행인수는 항상 현재 프로그람의 경로이름(C:\PDL\Chapt12\EX1221)이고 나머지 지령행인수들은 사용

자가 입력한것들이다. 지령행인수들은 공백문자에 의해 구분된다. 앞의 실례에서 지령행인수는 uno, dos, tres이다.

체계는 지령행인수를 기억기에 문자렬로 보관하고 이 문자렬들에로의 지적자배렬을 창조한다. 실례에서 그 배렬은 argv이다. 적당한 지적자에 의하여 개별적인 문자렬들을 호출할수 있다. 즉 첫 문자렬(경로이름)은 argv[0], 둘째 이름(이 실례에서 uno)은 argv[1], …이다.

실례 12-21은 인수들을 차례로 호출하고 지령행인수의 개수 argc를 상한으로 하는 for순환을 통하여 그것들을 출력한다.

main()에 인수로 주어지는 argc와 argv를 사용하는것이 관례로 되여있지만 다른 이름을 사용할수도 있다.

여기에 지령행인수의 다른 실례가 있다. 여기서는 지령행에서 사용자가 입력한 이름을 가지는 본문파일의 내용을 표시한다. 이와 같이 DOS지령 TYPE를 모의한다.

(실례 12-22) TYPE지령모의

```
#include <fstream>
#include <iostream>
using namespace std;
#include <process.h>
int main(int argc, char* argv[])
{
  if(argc != 2)
  {
    cerr << "\n형식화: OTYPE 파일이름";
    exit(-1);
  }
  char ch;
  ifstream inFile;
  inFile.open(argv[1]);
  if(!inFile)
  {
    cerr << "\n" << argv[1] << "파일을 열수 없습니다.";
    exit(1);
  }
  while(inFile.get(ch) != 0)
    cout << ch;
  return 0;
}
```

프로그람은 우선 사용자가 정확한 개수의 지령행인수를 입력하였는가를 검사한다. 실례 12-22와 같이 자체의 경로이름이 항상 첫 지령행인수로 된다. 둘째 인수는 표시하려는 파일의 이름이며 프로그람은 실행할 때 사용자가 입력해야 한다.

```
C:\>CTYPE ichar.cpp
```

그러므로 지형행인수의 총 개수는 2이다. 그렇지 않으면 사용자가 프로그람의 사

용방법을 모르므로 cerr를 통하여 오유통보를 내보낸다.

인수의 개수가 정확하면 프로그람은 이름이 제2지령행인수(argv[1])와 같은 파일을 열려고 시도한다. 파일을 열수 없으면 프로그람은 오유를 경고한다. 끝으로 while순환에서 파일을 한 문자씩 읽어들여 화면에 표시한다.

값이 0인 문자는 EOF를 경고한다. 이것은 EOF를 검사하는 다른 방법이다. 또한 앞에서 이미 고찰한것처럼 파일객체자체의 값을 사용할수도 있다.

```
while(inFile)
{
   inFile.get(ch);
   cout << ch;
}
```

또한 while순환을 cout << inFile.rdbuf();와 바꿀수도 있다.(실례 12-8)

제 10 절. 인쇄기출력

콘솔방식프로그람을 사용하여 인쇄기에 자료를 보내는것은 아주 간단하다. 하드웨어용의 특정파일이름들이 조작체계에 의해 이미 정의되여있으므로 장치를 파일처럼 취급할수 있다. 표 12-11에는 미리 정의된 이름을 보여준다.

표 12-11. 하드웨어이름

이 름	장 치
con	콘솔(건반과 영상표시장치)
aux 또는 com1	제1직렬포구
com2	제2직렬포구
prn 또는 lpt1	제1병렬인쇄기
lpt2	제2병렬인쇄기
lpt3	제3병렬인쇄기
nul	무효장치

대부분의 체계들에서 인쇄기는 첫째 병렬포구와 련결되므로 인쇄기를 위한 파일이름은 prn 또는 lptl이다.(체계가 다르게 구성된다면 적당한 이름으로 대신할수 있다.)

다음의 실례 12-23은 삽입연산자에 의하여 형식화된 출력방법으로 인쇄기에 문자렬과 수를 보낸다.

(실례 12-23) 인쇄기에로의 출력

```
#include <fstream>
using namespace std;
int main()
{
```

```
    char* s1 = "\n인쇄기에로 이 문자렬과 수값을 출력합니다: ";
    int n1 = 17982;
    ofstream outFile;
    outFile.open("PRN");
    outFile << s1 << n1 << endl;
    outFile << '\x0C';
    return 0;
  }
```

이러한 방법으로 임의의 량의 형식화된 출력을 인쇄기에 보낼수 있다.

'\x0C'문자는 인쇄용지를 바꾸게 한다.

다음의 실례 12-24에서는 지령행에 주어진 디스크파일의 내용을 인쇄기에 출력한다. 이 자료전송에도 한 문자씩 보내는 수법을 사용한다.

(실례 12-24) 인쇄지령모의

```
  #include <fstream>
  #include <iostream>
  using namespace std;
  #include <process.h>
  int main(int argc, char* argv[])
  {
    if(argc != 2)
    {
      cerr << "\n형식:OPRINT 파일이름";
      exit(-1);
    }
    char ch;
    ifstream inFile;
    inFile.open(argv[1]);
    if(!inFile)
    {
      cerr << "\n" << argv[1] << "파일을 열수 없습니다.";
      exit(1);
    }
    ofstream outFile;
    outFile.open("PRN");
    while(inFile.get(ch) != 0)
      outFile.put(ch);
    outFile.put('\x0C');
    return 0;
  }
```

자기의 .CPP원천파일과 같은 본문파일을 인쇄하는데 이 프로그람을 사용할수 있다. 이것은 DOS의 PRINT지령처럼 동작한다. 실례 12-22처럼 이 프로그람은 정확한 개수의 지령행인수를 입력하였는가 하는것과 지적된 파일을 성과적으로 열었는가를 검사한다.

요 약

이 장에서는 주로 스트림클라스의 계층구조를 시험하고 각종 입출력오유를 조종하는 방법을 보았다. 그다음 파일입출력을 처리하는 방법을 보았다. C++파일들은 여러 클라스의 객체들과 련결되는데 대체로 출력에 ofstream, 입력에 ifstream, 입출력에 fstream이 련결된다. 이 클라스 또는 그 기초클라스들의 성원함수들이 입출력조작에 사용된다. <<, put(), write()와 같은 연산자와 함수들은 출력에 쓰이고 >>, get(), read()는 입력에 쓰인다.

read()와 write()함수는 2진방식으로 작업하고 모든 객체는 그것이 포함하는 자료를 정렬하지 않고 디스크에 보관할수 있다. 하나의 객체를 보관할수 있다. 또한 많은 객체들로 이루어지는 배렬이나 자료구조체들도 보관할수 있다. 파일입출력을 성원함수에 의하여 조종할수 있다. 이것은 개별적객체가 응답능력을 가지게 하거나 혹은 클라스자체가 정적성원함수를 사용하여 입출력을 조종할수 있다.

오유조건검사는 파일을 조작할 때마다 매번 진행해야 한다. 파일객체자체는 오유가 발생할 때 값 0을 가진다. 또한 이때 성원함수들을 특별한 종류의 오유를 검사하는데 사용할수 있다. 발취연산자 >>와 삽입연산자 <<를 재정의하면 사용자정의자료형과 작업할수 있다. 기억기를 스트림처럼 고찰할수 있고 자료는 그것이 마치 파일인것처럼 거기에 전송할수 있다.

문 제

1. C++스트림은

① 함수를 통한 조종의 흐름이다.

② 한 곳으로부터 다른 곳에로의 자료의 흐름이다.

③ 특정클라스와 련결되여있다.

④ 파일이다.

어느것이 옳은가?

2. 대부분의 스트림클라스들의 기초클라스는 무엇인가?

3. 일반적으로 디스크입출력에 사용하는 스트림클라스 세개를 들어보시오.

4. ofstream클라스의 객체 saleFile을 창조하고 그것을 SALES.JUN이라는 파일과 련결하는 명령문을 쓰시오.

5. 입력스트림과 출력스트림은 각각 무엇에 대하여 작업을 하는가?

6. fooBar라는 ifstream객체가 파일끝에 이르렀는가 혹은 오유가 발생하였는가를 검사하는 if명령문을 쓰시오.

7. 삽입연산자 <<를 사용하여 ofstrem클라스의 객체에 본문을 출력할수 있는것은
① ofstream클라스가 스트림이기때문이다.
② 삽입연산자가 모든 클라스들과 작업하기때문이다.
③ 실제로 cout에 출력하고있기때문이다.
④ 삽입연산자가 ofstream에서 재정의되기때문이다.
어느것이 옳은가?
8. ofstream클라스의 fileOut객체에 한개 문자를 써넣는 명령문을 쓰시오.
9. ofstream형의 객체에 float형변수들을 포함하는 자료를 쓰기 위하여
① 삽입연산자
② seekg()
③ wrire()
④ put()를 사용하여야 한다. 어느것이 옳은가?
10. ifile이라는 ifstream객체의 내용을 buffer라는 배렬에 읽어들이는 명령문을 쓰시오.
11. app, ate와 같은 방식비트는
① ios클라스에서 정의된다.
② 파일이 읽기 혹은 써넣기용으로 여는가를 지정할수도 있다.
③ put()와 get()함수들과 작업한다.
④ 파일을 여는 방법을 지정한다.
어느것이 옳은가?
12. 현재위치가 파일들에 적용된다는것은 무엇을 의미하는가?
13. 파일지적자는 늘 파일의 주소를 포함하는가?
14. f1이라는 스트림객체안에서 현재위치를 13byte 이동하는 명령문을 쓰시오.
15. 명령문 f1.write((char*)&obj1, sizeof(obj1));은
① obj1의 성원함수들을 f1에 써넣는다.
② obj1의 자료를 f1에 써넣는다.
③ obj1의 성원함수와 자료를 f1에 써넣는다.
④ obj1의 주소를 f1에 써넣는다.
어느것이 옳은가?
16. 지령행인수는
① 지령재촉문에서 프로그람이름의 뒤에 입력된다.
② main()에로의 인수들을 통하여 호출된다.
③ 디스크파일로부터만 호출가능하다.
어느것이 옳은가?
17. skipws기발을 cin과 사용하면 어떤 현상이 일어나는가?

18. 지령행인수를 받아들일수 있는 main()의 선언자를 쓰시오.

19. 콘솔방식프로그람에서 인쇄기는 미리 정의된 어떤 파일이름을 사용하여 호출할수 있는가?

20. istream클라스의 객체들로부터 출력을 얻어서 클라스 Sample의 객체의 내용으로 화면에 표시하는 재정의된 >>연산자의 선언자를 쓰시오.

련습문제

1. 실례 6-7의 Distance클라스를 고찰하자. 실례 12-12와 비슷한 순환을 사용하여 사용자로부터 Distance값들을 얻어서 디스크파일에 써넣으시오. 파일안의 현존값들에 그것들을 추가하시오. 더는 입력할 값이 없다고 사용자가 경고할 때 그 파일을 읽어들여 값들을 모두 표시하시오.

2. 원천문자렬(임의의 .cpp파일)을 다른 파일에 복사하는 DOS의 COPY지령을 모의하는 프로그람을 쓰시오. 다음과 같이 두개의 지령행인수 즉 원천파일과 목적파일이름을 가지고 프로그람을 호출하시오.

```
C:>COPY srcfile.cpp destfile.cpp
```

프로그람에서 사용자가 정확한 수의 지령행인수를 입력했는가, 련관된 파일이 열렸는가를 검사하시오.

3. 지령행에 입력한 프로그람의 byte크기를 돌려주는 프로그람을 쓰시오.

```
C:>filesize program.txt
```

4. 순환에서 사용자에게 이름, 주소, 종업원 번호(unsigned long)로 이루어지는 이름자료를 입력하게 하시오. 다음 <<연산자에 의한 형식화된 입출력을 사용하여 이 세개의 항목들을 ofstream객체에 출력하시오. 문자렬들이 공백 혹은 다른 공백문자로 끝나야 한다. 사용자가 이름자료를 더는 입력하지 않는다고 응답했을 때 ofstream객체를 닫고 ifstream객체를 열고 파일의 자료를 모두 읽어서 표시하는 프로그람을 쓰시오.

5. 옹근수성원값 시, 분, 초를 가지는 Time클라스를 창조하시오. 성원함수 GetTime()은 사용자로부터 시간값을 읽어들이고 함수 PutTime()은 시간을 12:59:59 형식으로 표시한다. 오유를 최소로 줄이기 위하여 GetTime()함수에 오유검사기능을 추가하시오. 이 함수는 시, 분, 초를 따로따로 요구하여야 하며 매개에 대하여 ios오유상태기발과 값범위의 정확도를 검사하여야 한다. 시간은 0～23, 분과 초는 0～59이다. 이 값들은 문자렬로 입력하지 말고 옹근수로 직접 읽으시오. 이것은 실례 12-18과 같이 필요없이 소수점을 가진 입력을 화면에 표시할수 없다는것을 암시한다. main()에서 순환에 의하여 GetTime()에 의해 사용자로부터 시간값을 반복하여 얻어서 다음과 같이 PutTime()으로 표시하시오.

시간을 입력하시오: 1
분을 입력하시오: 10
초를 입력하시오: five
정확하지 않은 초를 입력하였습니다
초를 입력하시오: 5
시간은 1:10:5

6. 련습 4에서 Name이라는 클라스를 창조하시오. Name클라스에 디스크파일로부터 ofstream을 사용하여 객체의 자료를 써넣는 성원함수, ifstream을 사용하여 읽어들이는 함수를 정의하시오. <<와 >>에 형식화된 자료를 사용하시오. Read()와 Write()성원함수를 포함해야 한다. 그것들은 적당한 스트림을 열고 레코드를 읽어들이거나 써넣기 위한 명령문들을 포함해야 한다. Write()함수는 파일의 끝에 자료를 단순히 추가할 수 있다. Read()함수에는 읽으려는 레코드를 선택하는 방법이 요구된다. 한가지 방법은 레코드번호를 표시하는 파라메터를 사용하여 그것을 호출하는것이다.

일단 읽어들여야 할 레코드를 안다면 Read()함수가 레코드를 어떻게 찾겠는가?

seekg()함수를 사용할수 있지만 형식화된 입출력에서 레코드들은 모두 다른 길이(문자렬안의 문자수와 옹근수의 자리수에 따라서)를 가지므로 도움이 되지 못한다. 필요없는 레코드들을 뛰여넘어 읽으려는 레코드까지 가서 그것을 읽어야 한다. main()에서 이 성원함수를 호출하여 사용자가 여러개의 객체들에 대한 자료를 입력하고 파일에 써넣도록 하시오. 그다음 그 자료를 파일로부터 읽어서 표시하시오.

7. 파일스트림자체를 객체에 정적성원으로 만드는것은 객체에 파일스트림입출력을 추가하는 또 하나의 수법이다. 클라스의 개별적객체보다 총체적으로 그 클라스와 관련되여있는것처럼 스트림을 고찰하는것이 개념적으로 더 리해하기 쉽다. 또한 스트림을 한번만 열고 필요할 때 거기서 객체를 읽고 쓰는것이 더 효과적이다. 실례로 파일이 일단 열린 다음 읽기함수를 매번 호출하거나 파일에 다음 객체의 자료를 보낼수 있다. 읽어들이는 동안에 파일이 닫기지 않으므로 파일지적자는 파일을 자동적으로 항행한다. Name클라스의 정적자료항목으로써 fstream객체를 사용하도록 련습 4와 련습 6의 프로그람을 수정하시오. 그리고 우의 기능을 추가하시오. 이 스트림을 여는 정적함수와 파일의 선두로 파일지적자를 재설정하는 다른 정적함수를 쓰시오. 파일에 자료를 써넣고 파일로부터 모든 레코드를 읽어들일 때 재설정(Reset)함수를 사용할수 있다.

8. 실례 10-23을 고찰하자. 건을 눌러서 선택할수 있는 다음의 4가지 기능을 주는 프로그람을 작성하시오.

· 기억기의 목록에 련결을 추가하시오.(사용자는 한개 옹근수자료를 제공한다.)

· 기억기안의 모든 련결들의 자료를 표시하시오.

· 디스크파일에 모든 련결자료를 써넣으시오.(필요할 때 파일을 참고 또는 삭제한다.)

· 파일로부터 자료를 읽고 그것을 보관하기 위한 새로운 련결목록을 만드시오.

처음 두가지는 LinkList에 이미 실현된 성원함수들을 사용할수 있다. 디스크파일로부터 읽고 쓰기 위한 함수를 정의해야 한다. 읽기와 쓰기에 모두 같은 파일을 사용할수 있다. 파일은 자료만 보관해야 하고 지적자내용을 보관하지 않으며 목록을 읽어들일 때 련결이 없다.

9. 8장 련습 7에서 4기능분수수산기의 Fraction클라스에 재정의된 출력(<<) 및 입력(>>)연산자를 추가하시오. 연산자들은 식을 입력할 때 먼저 분수를 하나 얻고 다음 연산자를 얻으며 또한 분수를 얻어야 한다. 즉

```
cin >> frac1 >> op >> frac2;
```

10. 련습 9의 Fraction클라스의 입력(>>)연산자에 오유검사기능을 추가하시오. 오유검사에 의하여 첫 분수를 위한 재촉문, 그다음 연산자, 둘째 분수에 대한 재촉문을 내는것이 련습 9의 단일명령문보다 더 낫다. 이것은 오유통보와 결합될 때 더 융통성있다.

```
첫째 분수를 입력하시오: 5/0
나누는 수는 0일수 없습니다.
분수를 다시 입력하시오: 5/1
연산자(+,-,*,/)를 입력하시오: +
둘째 분수를 입력하시오: one third
입력오유입니다.
분수를 다시 입력하시오: 1/3
답 = 16/3
또 계산하겠습니까(y/n)? n
```

대화에서 알수 있는것처럼 ios오유기발과 나누는 수 0에 대하여 검사한다. 오유가 있으면 사용자가 분수를 다시 입력하게 하시오.

11. 11장의 련습 5에서 본 BMoney클라스를 고찰하시오. BMoney에 대하여 입출력하기 위한 출력(<<) 및 입력(>>)연산자를 추가하시오. 그리고 main()에서 그것을 시험하시오.

12. 실례 12-17에서 디스크파일의 종업원객체를 모두 검색하는 능력 즉 주어진 번호를 가지는 종업원을 찾는 기능을 추가하시오. 자료가 있으면 종업원의 자료를 표시해야 한다. 사용자는 f문자를 입력하는 방법으로 Find()함수를 호출하시오. 함수는 그다음 종업원번호에 대한 재촉문을 표시한다. 이때 함수는 정적이여야 하는가, 가상이여야 하는가? 검색과 표시조작은 기억기안의 자료를 변경하지 말아야 한다.

제 13 장. 여러 파일 프로그람작성

앞장들에서는 클라스선언, 성원함수, main()함수와 같은 C++프로그람의 각이한 부분들을 결합하는 방법을 보았다. 그러나 이 프로그람들은 하나의 파일로 구성되여있다. 그러면 여러 파일을 포함하는 경우에 프로그람조직방법을 고찰하기로 하자.

이 장에서는 여러 파일 프로그람들을 보여주는것과 함께 좀 더 길고 복잡한 응용프로그람들을 소개한다. 그 목적은 조작의 세부를 반드시 리해하려는데 있는것이 아니라 서로 련관된 대규모프로그람의 요소들을 어떻게 결합하는가하는 일반적리해를 얻으려는데 있다. 또한 이 프로그람들은 지금까지 본 간단한 실례들보다 더 실천적인 응용프로그람들에서 클라스를 사용하는 방법을 보여준다.

이 장에서는 우선 여러 파일 프로그람이 존재하는 리유와 여러 파일 프로그람의 창조에 대하여 설명하고 그 실례로서 매우 큰 수의 클라스를 포함하는 프로그람과 급수체계의 모의프로그람을 소개한다.

제 1 절. 여러 파일 프로그람이 존재하는 리유

여러 파일 프로그람을 사용하는 몇가지 리유가 있다. 그것은 클라스서고, 프로젝트에 대하여 작업하는 작성자들의 조, 프로그람의 개념설계와 관련된다.

1. 클라스서고

쏘프트웨어제작자들은 전통적인 수속지향언어들에 함수서고를 제공하기 위하여 오래동안 노력해왔다. 다른 프로그람작성자들은 자기가 쓴 루틴과 함수서고들을 결합하여 말단사용자용 응용프로그람을 창조한다. 서고는 광범한 분야의 함수들을 제공한다. 실례로 제작자는 통계계산을 조종하기 위한 함수서고 혹은 고급한 기억관리를 위한 함수들을 제공한다.

C++는 함수보다 클라스로 조직화되였으므로 C++프로그람용서고들이 클라스들로 이루어진다는것은 놀랄만한것이 못된다. 중요한것은 클라스서고가 함수서고보다 얼마나 우월한가 하는것이다. 클라스들은 자료와 함수들을 모두 밀봉하고 현실세계의 객체들을 더욱 근사하게 모형화하므로 클라스서고와 그것을 사용하는 응용프로그람은 함수서고가 제공해준것보다 훨씬 더 명백할수 있다.

그러므로 클라스서고는 전통적인 프로그람작성에서 함수서고가 수행하는것보다 C++프로그람작성에서 더 중요한 역할을 한다. 클라스서고는 프로그람작성에서 더 큰 몫을 차지할수 있다. 응용프로그람작성자는 클라스서고를 사용하면 최종산품을 창조하는데 적은 량의 프로그람작성만이 요구된다는것을 알게 된다. 또한 클라스서고를 더

많이 창조할수록 자기의 특정한 프로그람작성문제를 해결할수 있는 기회는 더욱 더 늘어난다.

제15장에서 표준형판서고의 주요 실례들을 고찰한다.

클라스서고는 보통 두개의 요소 즉 대면부와 실현부로 이루어진다.

그러면 대면부와 실현부의 차이는 무엇인가?

1) 대면부

클라스서고를 작성하는 사람을 클라스개발자, 서고를 사용하는 사람을 프로그람작성자라고 부르기로 하자.

프로그람작성자가 클라스서고를 사용하려면 클라스선언을 비롯한 각종 선언을 알아야 한다. 이 선언은 서고의 공개부와 같은것으로 생각할수 있으며 .H확장자를 가진 머리부파일이 원천코드형식으로 제공된다. 머리부파일은 #include지령에 의하여 의뢰자의 원천코드와 결합된다.

머리부파일에서 선언은 다음과 같은 리유에 의하여 공개되여야 한다.

즉 클라스의 서술을 읽는것보다 실제의 클라스정의를 보는것이 관례적이다. 여기서 중요한것은 프로그람작성자는 이 클라스들에 기초하여 객체들을 선언해야하고 이 객체로부터 성원함수들을 호출해야 한다. 이것은 원천파일안에서 클라스를 선언할 때에만 가능하다. 즉 사용자가 그것을 사용하므로 이 선언을 대면부(interface file)라고 한다. 프로그람작성자는 서고의 다른 부분을 알 필요가 없다.

2) 실현부

다른 한편 각종 클라스의 성원함수들의 내부작업을 프로그람작성자가 알아야 할 필요는 없다. 클라스개발자는 다른 쏘프트웨어개발자들처럼 보통 원천코드를 수정하거나 침해할수 있으므로 원천코드를 그대로 제공하지 않는다. 간단한 inline함수를 제외한 성원함수들은 목적(.OBJ)파일이나 서고(.LIB)파일들처럼 목적형식으로 배포된다. ActiveX와 COM과 같은 Windows에 고유한 클라스들과 그밖의 특수한 경우에 사용하기 위하여 다른 확장이 있을수 있다.

보통 클라스의 성원함수들의 정의를 포함하는 원천파일을 실현파일(implementatoin file)이라고 한다.

그림 13-1은 여러 파일체계안에서 각종 파일들이 어떻게 련결되는가를 보여준다.

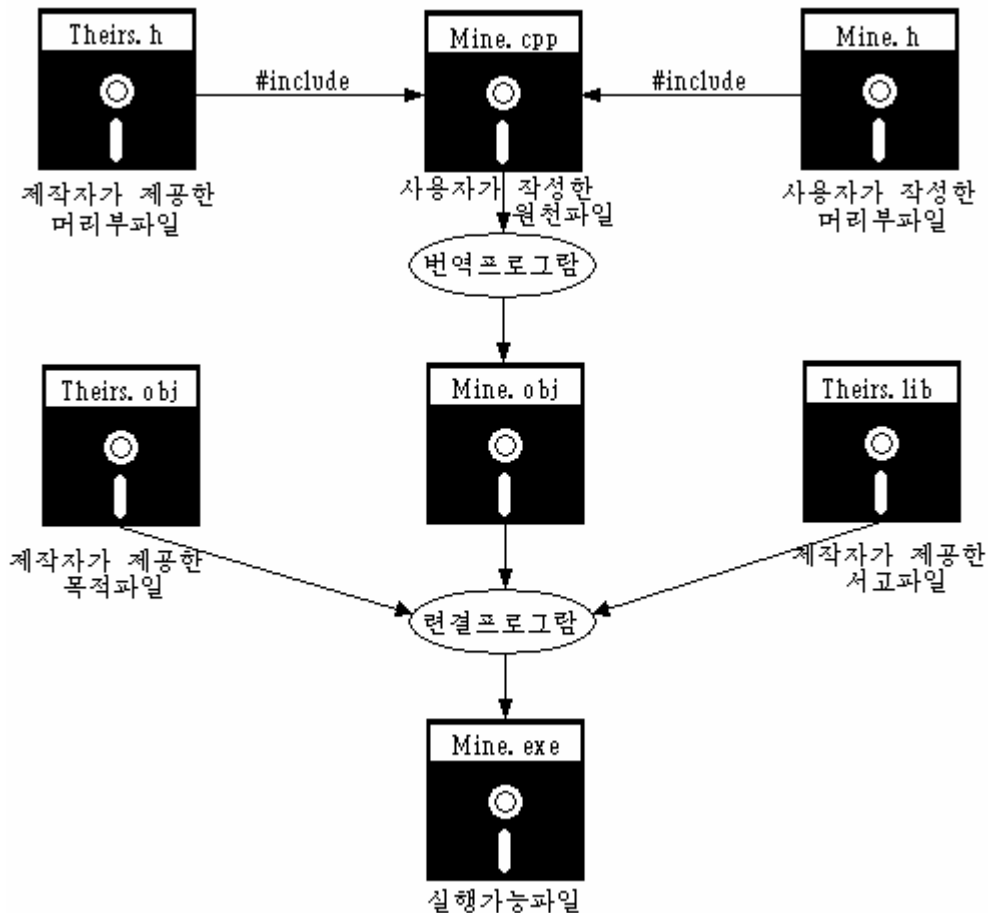



그림 13-1. 여러 파일응용프로그람에서의 파일들

이 장에서는 이 원리에 따라 조직화된 몇가지 큰 프로그람들을 보기로 한다. 첫째 프로그람은 매우 큰 수의 클라스를 소개한다. 매우 큰 수라는것은 자리수제한이 거의 없는 수를 말한다. 이러한 수들은 수학(즉 수천자리의 원주률계산 등)에서 매우 중요하다. 그리고 자체의 급수체계를 창조하는 클라스들을 제공한다. 핵반응로의 랭각체계와 같은 급수체계를 모의하기 위하여 발브, 탕크, 관, 그밖의 구성요소들을 결합할수 있다.

2. 조직과 개념화

프로그람은 클라스서고를 편의를 위해서가 아니라 다른 리유로 여러 파일로 나눌수 있다. 일반적으로 C와 같은 프로그람언어에서는 여러명의 프로그람작성자들이 참가하는 프로젝트를 포함한다. 매개 프로그람작성자의 부담을 개별적인 파일들로 고착시키는것은 프로젝트의 조직을 방조하고 프로그람의 각이한 부분들가운데서 대면부를 더 명백히 정의할수 있게 한다.

또한 프로그람을 그 기능에 따라서 여러 파일로 분류하는 경우가 가끔 있다. 실례

로 한개 파일은 영상표시장치에 표시하는 코드를 조종할수 있고 다른 파일은 수리해석을, 또 다른 파일은 디스크입출력을 조종할수 있다. 대규모프로그람에서 한개 파일은 관례상 조종가능한 범위까지 확장할수 있다.

여러 파일 프로그람들과 작업하는데 사용하는 수법들은 서로 비슷하고 프로그람을 분할하는 근거도 같다.

제 2 절. 여러 파일 프로그람의 창조

Theirs.OBJ라고 부르는 미리 작성된 클라스파일을 얻었다고 가정하자. (.LIB확장자를 가진 서고파일도 같은 방법으로 론의한다.) 이때 머리부파일 즉 Theirs.h가 제공된다. 또한 서고의 클라스들을 사용하여 자기의 프로그람 즉 원천파일 Mine.cpp을 작성하였다.

이 요소파일들 즉 Theirs.OBJ, Theirs.h, mine.cpp을 결합하여 하나의 실행프로그람을 만들려고 한다.

1. 머리부파일

머리부파일 Theirs.h는 #include지령에 의하여 자기의 원천파일에 간단히 포함된다.

```
#include "Theirs.h"
```

파일이름주위의 두점인용표는 번역프로그람에 기정머리부등록부가 아니라 먼저 현재등록부에서 파일을 찾게 한다.

2. 등록부

모든 요소파일 Theirs.OBJ, Thiers.h, Mine.cpp가 같은 등록부에 있다는것을 확인한다. 사실상 혼란을 피하기 위하여 프로젝트에 개별적인 등록부를 창조한다.

3. 프로젝트

대다수의 번역프로그람들은 프로젝트를 사용하여 여러개의 파일들을 관리한다. 프로젝트는 응용프로그람에 필요한 모든 파일을 포함한다. 또한 프로젝트파일이라고 부르는 특수파일에 이 파일들을 결합하기 위한 지령들이 들어있다. 프로젝트파일의 확장자는 번역프로그람제작회사에 따라 달라진다. 볼랜드에서는 .BP5, 마이크로쏘프트에서는 .DSP이다. 현대번역프로그람들은 이 파일을 자동적으로 구성하고 유지관리하므로 그것을 알 필요는 없다. 일반적으로 작성자는 번역프로그람에게 프로젝트에 추가하려는 원천파일(.cpp) 모두에 대하여 알려주어야 한다. 같은 방법으로 .OBJ와 .LIB파일들을 추가할수 있다. 머리부파일들은 번역프로그람마다 다르게 취급된다. 일부 번역프로

그람은 프로젝트에 그것을 추가할것을 요구한다. 다른 일부 번역프로그람은 원천파일이 #include등록부를 보고 자동적으로 추가한다. 모든 원천파일(.cpp와 .h)들을 번역하고 결과로 생기는 .OBJ파일들(다른 .OBJ 혹은 .LIB파일들)을 최종적인 .exe파일에 결합할데 대한 유일한 명령을 주어야 한다. 이것을 건설과정(build process)이라고 한다. 보통 .exe파일은 잘 실행된다.

프로젝트에 대한 하나의 요구는 매개 원천파일을 번역할 때 리력을 유지하는것이다. 마지막 건설이 진행된 후에 변경된 원천파일들만 다시 번역한다면 상당한 시간을 절약한다. 특히 대규모프로젝트에서 일부 번역프로그람은 make지령과 Build지령을 서로 구별한다. make는 마지막 건설후에 변경된 원천파일들만 번역하며 Build는 날자에 관계없이 모든 파일들을 번역한다.

제 3 절. 매우 긴 수 클라스

때때로 기본자료형 unsigned long은 일정한 옹근수산수연산에서 충분한 정확도를 주지 못한다. 표준 C++에서 unsigned long은 4,274,967,275까지의 옹근수를 유지하는 10자리의 가장 긴 옹근수이다. 이것은 휴대용콤퓨터에서 가능한 수이다. 그러나 이보다 더 많은 자리수를 가지는 옹근수와 작업해야 할 때 문제가 생긴다.

다음의 실례는 그 풀이를 제공한다. 이것은 1000자리까지의 옹근수를 유지하는 클라스를 제공한다. 더 긴 수를 만들려고 한다면 프로그람안의 한개 상수를 변경하면 된다.

1. 문자렬로서의 수

VeryLong클라스는 수를 수자들의 문자렬로 보관한다. 이러한 경우에 string클라스보다 작업하기 더 쉬운 char*형의 C문자렬이 있다. C문자렬의 사용은 긴 자리수의 연산능력을 제공한다. 즉 C++는 긴 C문자렬을 조종할수 있다. C문자렬은 단순히 배렬이다. 수를 C문자렬로 표시함으로써 그것을 요구대로 길게 만들수 있다. VeryLong에서는 두개의 자료성원 즉 수자들의 문자렬을 보관하는 char배렬과 문자렬의 길이를 의미하는 int가 있다.(이 길이는 반드시 필요하지 않지만 문자렬길이를 구하는데 strlen()을 반복 사용하여 보관한다.) 문자렬안의 수자들은 역순서로 즉 가장 낮은 자리수가 첫 원소 즉 vlstr[0]에 보관된다. 이것은 문자렬에 대한 여러가지 조작을 단순화한다. 그림 13-2는 문자렬로 보관된 수를 보여준다.

VeryLong수의 더하기와 곱하기를 위하여 사용자가 호출할수 있는 루틴을 준비한다.(덜기와 나누기루틴은 련습에 넘긴다.)

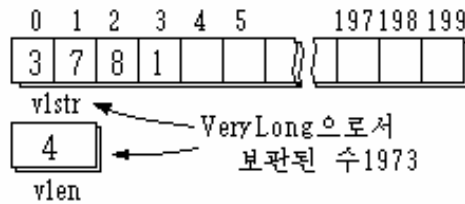



그림 13-2. VeryLong수

2. 클라스지정자

VeryLong의 머리부파일이 있다. 이것은 VeryLong클라스의 선언지정자를 보여준다.

```
// VeryLong.h
#include <iostream>
#include <cstring>
#include <stdlib.h>
using namespace std;
const int SZ = 1000;
class VeryLong
{
private:
   char vlstr[SZ];
   int vlen;
   VeryLong Mult10(const VeryLong) const;
   VeryLong MultiDigit(const int) const;

public:
   VeryLong operator*(const VeryLong);
   VeryLong operator+(const VeryLong);
   void GetVl();
   void PutVl() const;
   VeryLong(const unsigned long n);
   VeryLong(const char s[SZ]);
   VeryLong();
   virtual ~VeryLong();

};
```

VeryLong클라스에는 두개의 비공개성원함수가 있다, 하나는 VeryLong수와 하나의 수자를 곱한다. 다른 곱하기는 VeryLong수와 10을 곱한다. 이 루틴들은 곱하기루틴에 의하여 내부적으로 사용된다.

구성자가 세개 있다. 하나는 배렬의 선두에 null을 삽입하여 VeryLong을 0으로 설정하고 길이를 0으로 설정한다. 둘째 구성자는 그것을 역순서의 문자렬로 초기화하며 셋째 구성자는 long int값으로 초기화한다.

PutV1()성원함수는 VeryLong을 표시하고 GetV1()은 사용자로부터 VeryLong값

을 얻는다. 1000자리의 수자까지 입력할수 있다. 이 루틴에는 오유검사가 없다. 따라서 수자가 아닌 값을 입력하면 옳지 않은 결과가 나온다.

두개의 재정의된 연산자 +와 *는 더하기와 곱하기를 한다. 식

```
alpha  = beta * gamma + delta;
```

와 같은 VeryLong산수를 사용할수 있다.

3. 성원함수들

VeryLong.cpp파일은 성원함수정의를 보관한다.

```
// VeryLong.cpp: implementation of the VeryLong class.
#include "VeryLong.h"

VeryLong::VeryLong() : vlen(0)
{
  vlstr[0] = '\0';
}

VeryLong::~VeryLong()
{
}

VeryLong VeryLong::MultiDigit(const int d2) const
{
  char temp[SZ];
  int j, carry=0;
  for(j=0; j<vlen; j++)
  {
    int d1 = vlstr[j] - '0';
    int digitprod = d1 * d2;
    digitprod += carry;
    if(digitprod >= 10)
    {
        carry = digitprod / 10;
        digitprod -= carry * 10;
    }
    else
        carry = 0;
    temp[j] = digitprod + '0';
  }
  if(carry != 0)
    temp[j++] = carry + '0';
  temp[j] = '0';
  return VeryLong(temp);
}

VeryLong VeryLong::Mult10(const VeryLong v) const
{
```

```
    char temp[SZ];
    for(int j=v.vlen-1; j>=0; j--)
        temp[j + 1] = v.vlstr[j];
    temp[0] = '0';
    temp[v.vlen + 1] = '\0';
    return VeryLong(temp);
}

VeryLong::VeryLong(const char s[])
{
    strcpy(vlstr, s);
    vlen = strlen(s);
}

VeryLong::VeryLong(const unsigned long n)
{
    ltoa(n, vlstr, 10);
    strrev(vlstr);
    vlen = strlen(vlstr);
}

void VeryLong::PutVl() const
{
    char temp[SZ];
    strcpy(temp, vlstr);
    cout << strrev(temp);
}

void VeryLong::GetVl()
{
    cout << "Large Num? ";
    cin >> vlstr;
    vlen = strlen(vlstr);
    strrev(vlstr);
}

VeryLong VeryLong::operator +(const VeryLong v)
{
    char temp[SZ];
    int j;
    int maxlen = (vlen > v.vlen) ? vlen : v.vlen;
    int carry = 0;
    for(j=0; j<maxlen; j++)
    {
        int d1 = (j > vlen - 1) ? 0 : vlstr[j] - '0';
        int d2 = (j > v.vlen - 1) ? 0 : v.vlstr[j] - '0';
        int digitsum = d1 + d2 + carry;
        if(digitsum >= 10)
```

```
        {
            digitsum -= 10;
            carry = 1;
        }
        else
            carry = 0;
        temp[j] = digitsum + '0';
    }
    if(carry == 1)
        temp[j++] = '1';
    temp[j] = '\0';
    return VeryLong(temp);
}

VeryLong VeryLong::operator *(const VeryLong v)
{
    VeryLong pprod;
    VeryLong tempsum;
    for(int j=0; j<v.vlen; j++)
    {
        int digit = v.vlstr[j] - '0';
        pprod = MultiDigit(digit);
        for(int k=0; k<j; k++)
            pprod = Mult10(pprod);
        tempsum = tempsum + pprod;
    }
    return tempsum;
}
```

PutV1()과 GetV1()함수들은 C서고함수 strrev()에 의하여 C문자렬을 반전하여 보관하지만 수값을 표준대로 표시한다.

operator+()함수는 두개의 VeryLong을 더하며 셋째 VeryLong을 결과로 준다. 이것은 두개의 자리수를 하나씩 고찰함으로써 수행한다. 첨수 0위치의 두 수를 더하고 필요하다면 자리올림을 보관한다. 그다음 첨수 1위치의 수자들을 더하고 필요하다면 자리올림을 더한다. 이러한 방법으로 두 수의 더 큰자리수자들을 모두 더할 때까지 연산을 반복한다. 두 수값의 길이가 서로 다르면 짧은 수에 존재하지 않는 자리수를 더하기 전에 0으로 설정한다. 그림 13-3은 이것을 보여준다.

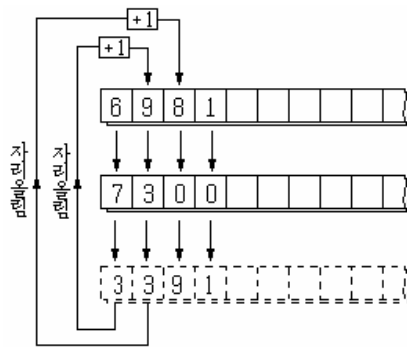



그림 13-3. VeryLong수들의 더하기

곱하기에는 operator*()함수를 사용한다. 이 함수는 곱하려는 수에 곱하는 수의 개별적수자들을 곱하는 방법으로 곱하기를 한다. 이때 MultiDigit()를 호출한다. 그다음 결과는 10의 적당한 배수를 곱하는 방법으로 수자의 위치가 일치하도록 자리밀기한다. 이때 Mult10()함수를 사용한다. 이 개별적계산의 결과들을 operator+()함수에 의하여 모두 더한다.

4. 응용프로그람

VeryLong클라스를 시험하기 위하여 실례 12-4를 변경하여 사용자가 입력한 수의 차례곱을 계산한다.

```
// V1App.cpp
#include <iostream>
using namespace std;
#include "VeryLong.h"

int main()
{
   unsigned long numb, j;
   VeryLong fact = 1;
   cout << "\n\nEnter Number: ";
   cin >> numb;
   for(j=numb; j>0; j--)
      fact = fact * j;
   cout << "Factorial = ";
   fact.PutVl();
   cout << endl;
   return 0;
}
```

이 프로그람에서 fact는 VeryLong변수이다. 다른 변수 numb와 j는 그리 크지 않으므로 VeryLong을 요구하지 않는다. 100의 차례곱을 계산하려면 numb와 j에는 3자

리수만 요구된다. 한편 fact는 158자리를 요구한다.

식 fact = fact * j;에서 long변수 j는 1인수구성자에 의하여 자동적으로 VeryLong으로 변환된 다음 곱하기가 수행된다. 100차례곱의 출력은 다음과 같다.

```
Enter Number: 100
Factorial=93326215443044152681697038806266700490711968284381821488592983895217599993229915608941463976156118286253697920827223758511852109168640000000000000000000000
```

제 4 절. 급수체계

살림집에 물이 어떻게 공급되는가? 또는 핵반응로의 랭각체계가 어떻게 동작하는가? 다음의 응용프로그람은 이 문제에 해답을 준다. 이 프로그람은 관, 발브, 탕크, 그밖의 요소들로 이루어지는 액체공급체계를 모형화한다. 이 실례는 주어진 경우에 일련의 클라스들을 창조하는 방법을 보여준다. 류사한 방법을 비행선조종에 사용되는 수력체계와 같은 다른 공정조종응용프로그람에도 사용할수 있다. 또한 급수체계나 화폐류통을 추적하는 경영체계에도 적용할수 있다.

그림 13-4는 산중턱에 건설한 작은 급수체계를 보여준다. 이 급수체계는 실제급수체계를 보여준다. 이 급수체계를 Pipes프로그람에서 모의한다.

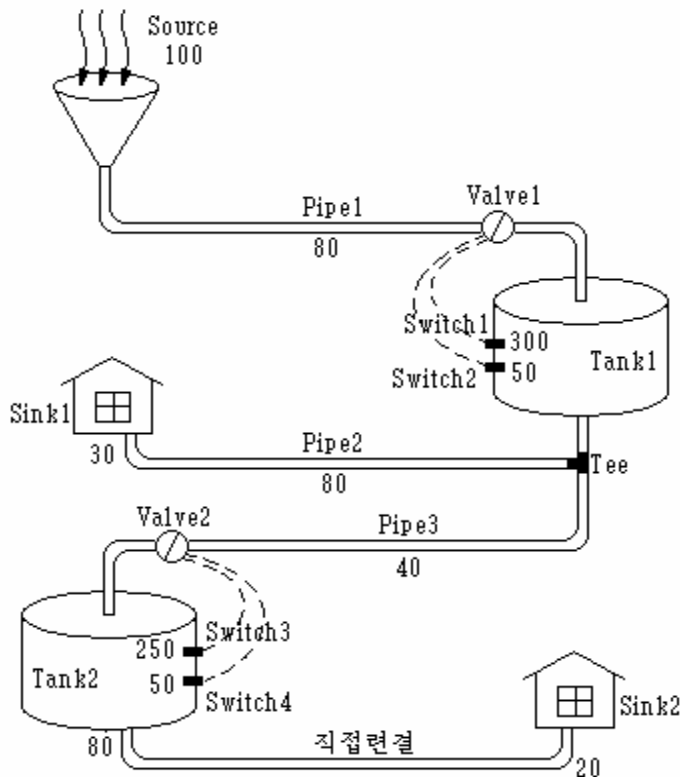



그림 13-4. 전형적인 급수체계

앞의 실례들처럼 이 프로그람을 세개의 파일로 나눈다. Pipes.h는 클라스선언을 포함하고 Pipes.cpp는 성원함수정의를 포함한다. 이 파일들은 클라스서고제작자에 의해 제공되는것으로 가정한다. PipesApp.cpp파일은 탕크, 발브, 관들이 일정하게 배치된 급수체계를 지정하기 위한것이다.

1. 급수체계의 구성요소

이 응용프로그람에서는 현실세계의 물리적객체를 찾는다. 이 객체들은 우리가 보고 수정할수 있는것으로서 프로그람의 객체와 밀접히 관련된다. 그러면 그 객체들은 무엇인가?

· 수원지(source)는 체계에 물을 공급한다. 현실세계에서 샘, 우물 또는 저수지와 대응된다. 물은 수원지로부터 항상 공급할수 있으나 일정한 고정속도보다 더 빨리 공급할수 없다.

· 소비지(sink)는 물의 사용지이다. 소비지는 살림집, 공장, 농장 또는 물소비지들의 그룹으로 표시된다. 소비지는 체계로부터 고정속도로 공급되는 물을 소비한다.

· 관(pipe)은 물을 일정한 거리에 나른다. 관은 통과할수 있는 물량을 제한하는 특성을 가지고있다. 관으로 흘러들어오는 물량은 그로부터 흘러나가는 물량과 같다.

· 탕크(tank)는 물을 저장한다. 또한 입출구흐름을 완화시킨다. 탕크로 들어오는 물의 속도는 그것이 흘러나오는 속도와 다를수 있다. 실제로 입구흐름이 출구흐름보다 더 크면 탕크의 물량은 늘어난다. 탕크는 그 내부체적에 의하여 결정되는 고유한 최대 출구흐름속도를 가진다.

· 탕크에 물이 넘쳐나지 않게 하기 위하여, 혹은 물이 다 빠지지 않게 하기 위하여 탕크에 여닫이(switch)를 련결할수 있다. 여닫이는 탕크의 물량이 일정한 량에 이를 때 투입한다. 여닫이는 보통 발브를 동작시키는데 사용하며 탕크의 물의 높이를 조종한다.

· 발브(valve)는 물의 흐름을 조절한다. 흐름에 저항이 없도록 그것을 열고 또한 흐름을 완전히 멈추기 위하여 발브를 닫을수 있다. 발브는 사보기구에 의하여 동작하며 탕크와 련결된 여닫이에 의하여 조종된다.

2. 흐름, 수압과 역수압

체계의 매개 요소는 세개의 중요한 개념 즉 흐름, 수압, 역수압을 가진다. 한 요소를 다른 요소와 련결할 때 이것들을 결합한다.

1) 흐름

급수체계의 제일 기초에 놓여있는 특성은 흐름이다. 흐름은 단위시간당 요소를 통과하는 물의 량으로서 체계를 모형화할 때 측정에 사용된다.

대체로 한 요소로 들어오는 흐름은 나가는 흐름과 같다. 이것은 관과 발브인 경우이며 탕크인 경우에는 그렇지 않다.

2) 수압

흐름이 모든것을 결정하지 못한다. 실례로 발브를 닫았을 때 거기에 들어오는 흐름과 나가는 흐름은 둘다 정지하지만 물은 여전히 발브를 지나려고 한다. 흐름에 대한 이 포텐샬을 수압이라고 한다. 수원지의 모든 탕크는 일정한 수압으로 물을 공급한다. 체계의 여유를 허용한다면 이 수압은 비례흐름이 생기게 한다. 즉 수압이 클수록 흐름도 많다. 그러나 발브를 닫으면 흐름은 수압에 관계없이 멎는다. 수압은 흐름처럼 한 요소에서 다른 요소로 내리흐름으로 전달된다.

탕크는 표식은 물론 수압을 가진다. 탕크로부터 내리흐르는 수압은 올리흐름압력이 아니라 탕크에 의해 결정된다.

3) 역수압

수압과 반대로 작용하는것이 역수압이다. 이것은 요소들의 흐름에 대한 저항에 의해 생긴다. 실례로 직경이 작은 관은 물의 흐름이 떠지게 하며 그것은 수압이 얼마인가 하는데 문제가 없다. 역수압은 그것을 일으키는 요소와 올리흐르는 모든 요소들로 들어오는 속도를 감소시킨다.

역수압은 흐름 및 수압과 반대방향으로 향한다. 그것은 내리흐름요소로부터 올리흐름요소에로 전달된다. 탕크에는 수압과 흐름은 물론 역수압도 작용한다.

3. 요소의 입구와 출구

대체로 흐름, 수압, 역수압은 요소의 량끝에서 같다. 실례로 관의 한끝으로 들어오는 흐름은 다른 끝으로 나가는 흐름과 같다. 그러나 이 특성은 또한 올리흐름과 내리흐름측에 따라 다르다. 발브를 닫을 때 그 내리흐름측에 대한 수압은 0으로 되며 올리흐름측에 대한 입구가 무엇인가에는 문제가 없다. 탕크로 들어오는 흐름은 나가는 흐름과 다를수 있다. 관의 출구수압은 관의 저항으로 인하여 입구수압보다 작을수 있다.

따라서 매개 요소가 임의로 주어진 경우에 6개의 값으로 특징지을수 있다.

세개의 입구 즉 수압(올리흐름요소로부터), 역수압(내리흐름요소로부터), 흐름(올리흐름으로부터), 또한 세개의 출구 즉 수압(내리흐름요소에 대하여), 역수압(올리흐름요소에 대하여), 흐름(내리흐름요소로부터)이 있다. 이 상황은 그림 13-5와 같다.

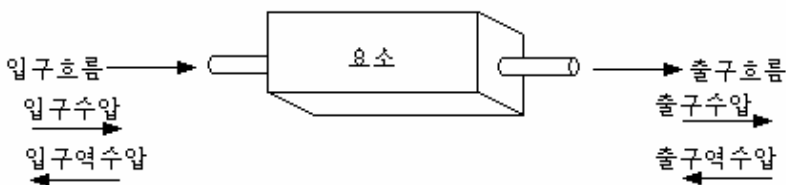



그림 13-5. 요소의 특성

요소의 출구는 그 입구로부터 계산되며 또한 내부특성과 요소의 상태(즉 관의 저항 혹은 발브를 열었는가, 닫았는가)로부터 계산된다. 매개 요소의 성원함수 Tick()는 고정시격으로 박자를 발생시키며 입구와 내부특성에 기초한 요소의 출구를 계산하는데 사용된다. 례를 들어 관에서는 입구수압이 높아지면 흐름은 그에 따라서 늘어난다.

4. 결합

급수체계를 작성하려면 세가지 요소들을 모두 결합하여야 한다. 한 요소를 다른 요소와 결합하여 물이 한 요소로부터 다른 요소로 흐르게 할수 있다.(이런 방법으로 여닫이를 결합하지 않는다. 즉 그것이 물을 나르지 않기때문이다.) 흐름과 함께 수압과 역수압을 결합하여야 한다. 그것은 역시 한 요소로부터 다른 요소로 전달되기때문이다.

이와 같이 결합한다는것은 올리흐름객체로부터의 출구수압과 출구흐름, 내리흐름객체의 입구수압과 입구흐름을 설정한다는것과 내리흐름객체로부터의 출구역수압을 올리흐름객체의 입구역수압으로 설정한다는것을 의미한다. 이것을 그림 13-6에 보여준다.

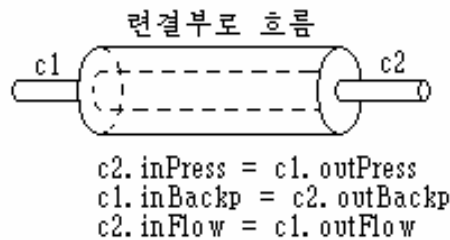



그림 13-6. 요소들사이의 결합

5. 가설의 단순화

복잡한 수값계산을 피하기 위하여 가설을 단순화한다.

프로그람에서 역수압을 적당히 흐름의 안정(ease of flow)이라고 부른다.

우리가 이 특성에 사용하는 값은 결과흐름에 비례하여 적은 량이 흐를 때 작아지고 큰 량이 흐를 때 커진다. 실제역수압은 결과흐름과 호상 련관될수 있으나 프로그람을 매우 복잡하게 만든다.

수압과 역수압은 흐름과 같은 단위로 측정되는것으로 가정한다. 흐름을 계산하기 위하여 체계에 들어오는 물이 흐르는 수압과 그 흐름을 방해하는 역수압을 시험한다. 결과흐름은 이 두 수값의 최소값이다. 따라서 수원지로부터 0.1㎥/min으로 공급된다면 관은 0.06㎥/min의 저항을 가지며 이것은 0.06㎥/min의 역수압을 발생시키고 흐름은 0.06㎥/min으로 된다.

이 가설은 수압흐름의 현실세계를 정확히 모형화하지 못하며 현실세계에서 흐름은

수압과 역수압이 련관된 복잡한 공식들에 의하여 결정된다. 그렇지만 이것은 좋은 첫 근사값을 제공한다.

탕크의 출구수압이 상수라고 가정하자. 실제로 그것은 탕크의 용량에 의존한다. 그러나 소비지보다 매우 높은 곳에 있는 탕크에서 이것은 합리적인 근사값이다.

이 문제를 해결하는데서 피할수 없는 내부종류의 불완전성이 있다.

우리가 모의하고있는 급수체계는 시간에 따라 련속 변하는 상사체계이다. 그러나 우리의 모형은 수자모형이다. 따라서 일정한 박자 혹은 시간간격으로 요소와 상태를 모형화한다. 그러므로 어떤것이 변경될 때(실례로 발브를 열 때) 결과수압에 대하여 여러 순환주기를 가질수 있으며 흐름변경이 체계를 통하여 전달된다.

이 림시적인 상태는 체계의 동작을 분석하는데서 무시될수 있다.

6. 프로그람설계

이 프로그람의 목적은 서로 다른 급수체계를 모형화하기 위한 클라스계층을 창조하는것이다. 응용프로그람에서 클라스들이 무엇을 표시하는가를 리해하기 쉽다. 매개 종류의 요소에 대하여 클라스를 창조한다. 즉 Valve클라스, Tank클라스, Pipe클라스 등 이 클라스들이 정의되면 프로그람작성자는 특정체계를 모형화하는데 필요한 요소들을 련결할수 있다.

Pipes.h파일은 쏘프트웨어제작자가 제공한다.

```
// Pipes.h
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;

const int INFINITY = 32767; // reverse pressure
enum OFFON { OFF, ON };

class Tank;
class Component
{
protected:
   int inPress, outPress;
   int inBackp, outBackp;
   int inFlow, outFlow;

public:
   Component() : inPress(0), outPress(0), inBackp(0), outBackp(0),
          inFlow(0), outFlow(0) {}
   virtual ~Component() {}
   int Flow() const { return inFlow; }
   void operator >= (Component&);
```

```
    friend void Tee(Component&, Component&, Component&);
};

class Source : public Component
{
public:
    Source(int outp) { outPress = inPress = outp; }
    void Tick();
};

class Sink : public Component
{
public:
    Sink(const int obp) { outBackp = inBackp = obp; }
    void Tick();
};

class Pipe : public Component
{
private:
    int resist;

public:
    Pipe(const int r) { inBackp = resist = r; }
    void Tick();
};

class Valve : public Component
{
private:
    OFFON status;

public:
    Valve(const OFFON s) { status = s; }
    OFFON& Status() { return status; }
    void Tick();
};

class Tank : public Component
{
private:
    int contents;
    int maxOutPress;

public:
    Tank(const int mop) { maxOutPress = mop; contents = 0; }
    int Contents() const { return contents; }
    void Tick();
```

```
};

class Switch
{
private:
  OFFON status;
  int cap;
  Tank* tankPtr;

public:
  Switch(Tank* tptr, const int tcap)
     { tankPtr = tptr; cap = tcap; status = OFF; }
  int Status() const { return status; }
  void Tick() { status = (tankPtr->Contents() > cap) ? ON : OFF; }
};
```

Pipes.cpp파일은 클라스성원함수의 정의를 포함한다. 이것은 클라스제작자가 제공한다.

```
// Pipes.cpp
#include "Pipes.h"

void Component::operator >= (Component& c2)
{
  c2.inPress = outPress;
  inBackp = c2.outBackp;
  c2.inFlow = outFlow;
}

void Tee(Component& src, Component& c1, Component& c2)
{
  if((c1.outBackp == 0 && c2.outBackp == 0) ||
                      (c1.outBackp == 0 && c2.outBackp == 0))
   {
     c1.inPress = c2.inPress = 0;
     src.inBackp = 0;
     c1.inFlow = c2.inFlow = 0;
     return;
   }
  float f1 = (float)c1.outBackp / (c1.outBackp + c2.outBackp);
  float f2 = (float)c2.outBackp / (c1.outBackp + c2.outBackp);
  c1.inPress = src.outPress * f1;
  c2.inPress = src.outPress * f2;
  src.inBackp = c1.outBackp + c2.outBackp;
  c1.inFlow = src.outFlow * f1;
  c2.inFlow = src.outFlow * f2;
}

void Source::Tick()
```

```
{
   outBackp = inBackp;
   outFlow = (outPress < outBackp) ? outPress : outBackp;
   inFlow = outFlow;
}

void Sink::Tick()
{
   outPress = inPress;
   outFlow = (outBackp < outPress) ? outBackp : outPress;
   inFlow = outFlow;
}

void Pipe::Tick()
{
   outPress = (inPress < resist) ? inPress : resist;
   outBackp = (inBackp < resist) ? inBackp : resist;
   if(outPress < outBackp && outPress < resist)
      outFlow = outPress;
   else if(outBackp < outPress && outBackp < resist)
      outFlow = outBackp;
   else
      outFlow = resist;
}

void Valve::Tick()
{
   if(status == ON)
   {
      outPress = inPress;
      outBackp = inBackp;
      outFlow = (outPress < outBackp) ? outPress : outBackp;
   }
   else
   {
      outPress = 0;
      outBackp = 0;
      outFlow = 0;
   }
}

void Tank::Tick()
{
   outBackp = INFINITY;
   if(contents > 0)
   {
      outPress = (maxOutPress < inBackp) ? maxOutPress : inBackp;
      outFlow = outPress;
```

```
    }
    else
    {
        outPress = 0;
        outFlow = 0;
    }
    contents += inFlow - outFlow;
}
```

7. 결합부프로그람작성

프로그람의 기본부분은 간단하며 프로그람에서 결합부들을 서술하는 직관적인 방법이다. 다음과 같이 함수를 사용할수 있다.

```
Connect(valve1, tank1);
```

그러나 이것은 혼돈할수 있다. 즉 올리흐름요소가 오른쪽 인수인가, 왼쪽 인수인가?

더 좋은 방법은 요소들사이의 결합부를 포함하도록 연산자를 재정의하는것이다. 우리는 >=연산자를 선택하고 왼쪽에서 오른쪽에로 흐름방향을 간접적으로 제한한다. 이것을 흐름(flows-into)연산자라고 부른다. 프로그람명령문은 다음과 같이 결합을 수행한다.

```
valve1 >= tank1;
```

이것은 valve1로부터 tank1에로의 흐름을 의미한다.

8. 기초와 파생클라스

프로그람을 실행할 때 우선 여러 객체들중에서 류사성을 찾는다. 공통특성은 기초클라스에 놓을수 있고 요소들을 구별하는 개별적특성들은 파생클라스에 놓는다.

9. Component기초클라스

응용프로그람에서 여닫이를 제외한 객체는 그것을 지나는 물흐름을 가지며 서로 련결될수 있다. 그러므로 결합을 허용하는 기초클라스를 창조한다. 그것을 Component(부분품)라고 한다.

```
class Component
{
protected:
    int inPress, outPress;
    int inBackp, outBackp;
    int inFlow, outFlow;

public:
    Component() : inPress(0), outPress(0), inBackp(0), outBackp(0),
                  inFlow(0), outFlow(0) {}
    virtual ~Component() {}
```

```
        int Flow() const { return inFlow; }
        void operator >= (Component&);
        friend void Tee(Component&, Component&, Component&);
    };
```

Component는 수압, 역수압, 흐름을 가진다. 이것들은 모두 두개의 값 즉 요소에로의 입구와 요소로부터 출구를 가진다. 입구에서는 올리흐름으로부터 객체에로의 흐름, 그 올리흐름측에서 객체들에 의해 발생된 수압, 내리흐름측에서 객체들에 의해 발생되는 역수압을 가진다. 출구에서는 객체밖으로의 흐름, 내리흐름객체에로 전달되는 수압, 올리흐름객체에 전달되는 역수압이 있다. 이 값들은 모두 Component클라스의 객체에 보관된다. 이 클라스의 구성자는 모든 자료항목을 0으로 초기화하며 다른 성원함수는 대다수 부분품에 대하여 체계가 어떻게 작업하는가를 보여주는 흐름량을 돌려준다.

10. 흐름연산자

흐름연산자 >=는 올리흐름요소를 내리흐름요소와 결합한다. 세개의 입구(내리흐름객체의 수압과 흐름, 올리흐름객체의 역수압)는 세개의 출구(올리흐름객체의 수압과 흐름, 내리흐름객체의 역수압)와 같이 설정한다.

```
void Component::operator >= (Component& c2)
{
    c2.inPress = outPress;
    inBackp = c2.outBackp;
    c2.inFlow = outFlow;
}
```

>=연산자는 Component클라스의 성원함수로 정의한다. 또한 성원함수로 정의할수도 있다. 다른 종류의 결합 Tee()함수는 동료이여야 한다.

>=연산자는 기초클라스 Component의 객체들에 적용되므로 Tank, Valve, Pipe와 같은 파생클라스의 객체들에 대해서도 동작한다. 이것은 각종 결합을 조종하기 위하여 따로따로 함수를 쓰지 않게 해준다. 즉

```
friend void operator>=(Pipe&, Valve&);
friend void operator>=(Valve&, Tank&);
friend void operator>=(Tank&, Sink&);
```

11. 파생클라스들

체계의 물리객체를 모의하는 클라스들은 기초클라스 Component로부터 파생된다. 이것들은 Source, Sink, Pipe, Valve, Tank로서 매개가 고유한 특성을 가진다. Source는 고정입구수압을 가지고 Sink는 고정역수압을 가진다. Pipe는 고정시격저항을 가지며 그 출구역수압은 고정값보다 클수 없다. Valve는 OFF/ON형의 상태이며 기정으로 OFF를 가진다. Tank는 용량을 가진다. 발브의 상태와 탕크용량은 프로그람을 실행할때 반영된다.

일반적으로 프로그람에서 상수인 변수들(관의 저항, 탕크의 출구수압)은 객체들이

처음으로 창조될 때 초기화된다.

모든 파생클라스와 Switch클라스는 Tick()라는 함수를 가지고있다. Tick()함수는 시간주기별로 체계의 매개 객체를 호출하여 객체의 내부상태를 갱신하고 세개의 입구와 세개의 출구(수압, 역수압, 흐름)를 계산한다.

— Tee()함수

Tee()함수는 하나의 입구흐름과 두개의 출구흐름으로 나눈다. 매개의 내리흐름요소로 가는 흐름은 매개 요소의 역수압에 비례한다. 저항이 큰 관은 저항이 작은것보다 더 작은 비률의 흐름을 얻는다.

Tee()함수는 세개의 인수 즉 원천지요소와 두개의 내리흐름요소를 가지고 호출한다.

```
Tee(input, output1, output2);
```

이것은 세개의 파라메터를 가지며 세개 요소를 결합하는 함수이다. 연산자를 사용하는것이 더 좋을수 있다. 즉

```
input >= output1 + output2;
```

그러나 C++에는 재정의할수 있는 3항연산자가 없다.

12. Switch클라스

Switch클라스는 Tank클라스와 특별한 관계를 가진다. 매개 탕크는 두개의 여닫이와 련결된다. 하나의 여닫이는 탕크수위가 일정한 최소량에 이르렀을 때(탕크가 거의 비였을 때) 투입하기 위해 설정된다. 다른 여닫이는 탕크수위가 일정한 최대값으로 될 때(탕크가 다 찼을 때) 투입된다. 그 최대값이 탕크의 용량을 결정한다.

여닫이와 탕크사이의 관계 즉 여닫이는 탕크에 의하여 소유되는것으로 정의한다. 여닫이가 정의될 때 두개의 값이 주어진다. 하나는 그것을 소유하는 탕크의 주소이고 다른 하나는 그것이 투입되는 탕크의 물높이이다. Switch의 Tick()성원함수는 탕크용량을 직접 호출하기 위하여 그 탕크의 주소를 사용한다. 이것은 그것이 투입되는가, 차단되는가를 결정하는 방법이다.

여닫이는 탕크로부터 나오는 물의 흐름을 조절하는 발브를 조종하는데 사용된다. 탕크가 다 차면 발브를 차단하고 거의 비게 되면 발브를 다시 투입한다.

13. PipeApp.cpp파일

프로그람의 main()부분은 응용프로그람작성자에 의하여 급수체계를 모의하기 위하여 작성된다. 여기에 PipeApp.cpp의 프로그람이 있다. 이 파일은 main()함수만 포함한다.

```
// PipeApp.cpp
#include "Pipes.h"

int main()
```

```
{
  char ch = 'a';
  Source src(100);
  Pipe pipe1(80);
  Valve valve1(ON);

  Tank tank1(60);
  Switch switch1(&tank1, 300);
  Switch switch2(&tank1, 50);

  Pipe pipe2(80);
  Sink sink1(30);
  Pipe pipe3(40);
  Valve valve2(ON);

  Tank tank2(80);
  Switch switch3(&tank2, 250);
  Switch switch4(&tank2, 50);
  Sink sink2(20);
  cout<< "새로운 시간박자에 대하여 Enter건을 누르고 완료하려면 'x'를 누르시오.\n";
  while(ch != 'x')
  {
     src >= pipe1;
     pipe1 >= valve1;
     valve1 >= tank1;
     Tee(tank1, pipe2, pipe3);
     pipe2 >= sink1;
     pipe3 >= valve2;
     valve2 >= tank2;
     tank2 >= sink2;
     src.Tick();
     pipe1.Tick();
     valve1.Tick();
     tank1.Tick();
     switch1.Tick();
     switch2.Tick();
     pipe2.Tick();
     sink1.Tick();
     pipe3.Tick();
     valve2.Tick();
     tank2.Tick();
     switch3.Tick();
     switch4.Tick();
     sink2.Tick();

     if(valve1.Status() == ON && switch1.Status() == ON)
         valve1.Status() = OFF;
     if(valve1.Status() == OFF && switch2.Status() == OFF)
```

```
        valve1.Status() = OFF;
    if(valve2.Status() == ON && switch3.Status() == ON)
        valve2.Status() = OFF;
    if(valve2.Status() == OFF && switch4.Status() == OFF)
        valve2.Status() = ON;
    cout << "Src=" << setw(2) << src.Flow();
    cout << " P1=" << setw(2) << pipe1.Flow();

    if(valve1.Status() == OFF)
        cout << " V1=OFF";
    else
        cout << " V1=ON";
    cout << " T1=" << setw(3) << tank1.Contents();
    cout << " P2=" << setw(2) << pipe2.Flow();
    cout << " Sink1=" << setw(2) << sink1.Flow();
    cout << " P3=" << setw(2) << pipe3.Flow();

    if(valve2.Status() == OFF)
        cout << " V2=OFF";
    else
        cout << " V2=ON";
    cout << " T2=" << setw(3) << tank2.Contents();
    cout << " Sink2=" << setw(2) << sink2.Flow();
    ch = getch();
    cout << '\n';
  }
  return 0;
}
```

1) 요소의 선언

main()에서는 우선 여러가지 요소들 즉 관, 발브, 탕크들이 선언된다. 이때 그 고정특성들도 초기화된다. 관에는 고정저항이 주어지고 탕크용량은 빈것으로 초기화된다.

2) 련결과 갱신

main()의 작업은 순환으로 되여있다. 순환이 진행되는 매 시간은 한시간주기 혹은 한박자로 표시된다. Enter건을 누르면 새 주기가 시작되고 프로그람의 사용자는 체계의 박자처럼 동작한다. 순환을 완료하고 프로그람을 끝내려면 'x'건을 누른다.

순환에서 첫 일감은 매개 요소들을 련결하는것이다. 수원지 src는 pipe1과 련결되고 pipe1은 valve1과 련결되며 따라서 결과의 체계는 그림 13-4와 같아진다.

일단 련결되면 모든 요소들의 내부상태는 거의 Tick()함수에 의하여 갱신된다.

발브는 그 이전상태와 여닫이에 기초하여 if명령문에서 열리고 닫긴다. 목적은 적당히 발브를 열거나 닫음으로써 웃여닫이와 아래여닫이사이에서 탕크의 용량을 유지하는데 있다.

탕크용량이 웃쪽 여닫이에 이르면 이 여닫이가 투입되고 if명령문은 발브를 닫게

한다.용량이 아래여닫이까지 내려가면 여닫이를 차단하여 발브를 연다.

3) 출력

체계에서 어떤 현상이 일어나는가를 보기 위하여 cout명령문을 사용하여 여러가지 요소들의 흐름, 탕크용량, 시간에 따라 변하는 값들의 상태를 표시할수 있다. 그림 13-7은 이것을 보여준다.

```
새로운 시간박자에 대하여 Enter를 누르고 완료하려면 'x'를 누르시오.
Src= 0 P1= 0 V1=ON T1=  0 P2= 0 Sink1= 0 P3= 0 V2=ON T2=  0 Sink2= 0
Src= 0 P1=' 0 V1=ON T1=  0 P2= 0 Sink1= 0 P3= 0 V2=ON T2=  0 Sink2= 0
Src= 0 P1= 0 V1=ON T1= 80 P2= 0 Sink1= 0 P3= 0 V2=ON T2=  0 Sink2= 0
Src=80 P1= 0 V1=ON T1=100 P2= 0 Sink1= 0 P3= 0 V2=ON T2=  0 Sink2= 0
Src=80 P1=80 V1=ON T1=120 P2=25 Sink1= 0 P3=34 V2=ON T2=  0 Sink2= 0
Src=80 P1=80 V1=ON T1=140 P2=25 Sink1=25 P3=34 V2=ON T2=  0 Sink2= 0
Src=80 P1=80 V1=ON T1=160 P2=25 Sink1=25 P3=34 V2=ON T2= 34 Sink2= 0
Src=80 P1=80 V1=ON T1=180 P2=25 Sink1=25 P3=34 V2=ON T2= 48 Sink2= 0
Src=80 P1=80 V1=ON T1=200 P2=25 Sink1=25 P3=34 V2=ON T2= 62 Sink2=20
Src=80 P1=80 V1=ON T1=220 P2=25 Sink1=25 P3=34 V2=ON T2= 76 Sink2=20
Src=80 P1=80 V1=ON T1=240 P2=25 Sink1=25 P3=34 V2=ON T2= 90 Sink2=20
Src=80 P1=80 V1=ON T1=260 P2=25 Sink1=25 P3=34 V2=ON T2=104 Sink2=20
Src=80 P1=80 V1=ON T1=280 P2=25 Sink1=25 P3=34 V2=ON T2=118 Sink2=20
Src=80 P1=80 V1=ON T1=300 P2=25 Sink1=25 P3=34 V2=ON T2=132 Sink2=20
Src=80 P1=80 V1=OFF T1=320 P2=25 Sink1=25 P3=34 V2=ON T2=146 Sink2=20
Src=80 P1=80 V1=OFF T1=340 P2=25 Sink1=25 P3=34 V2=ON T2=160 Sink2=20
Src=80 P1=80 V1=OFF T1=280 P2=25 Sink1=25 P3=34 V2=ON T2=174 Sink2=20
Press any key to continue_
```

그림 13-7. Pipes의 출력

그림에서 일부 값들이 우연히 0으로 된다. 이것은 이미 언급한 모의의 수자적특성 때문이다. 이것은 무시할수 있다.

대다수 급수체계에서 목표는 여러개의 소비지에 련속적으로 물을 공급하는것이다. 실제의 출력은 모의체계에 일부 문제가 있음을 보여준다. sink1의 흐름을 25와 32ℓ/min으로 tank2가 찼는가 차지 않았는가에 따라 엇바꾼다. 급수체계의 사용자는 일정하게 공급하는것을 더 좋아한다. 더우기 sink2는 전혀 흐름이 없는 주기를 포함한다. 이 결함을 없애기 위하여 체계의 일부 요소들을 다시 조절할 필요가 있다.

물론 cout는 매우 복잡하지 않은 출력체계를 제공한다. 요소들의 그림을 화면에 표시하는 그라프출력을 제공하는것은 힘들다. 매개 요소에 짜넣은 Display()함수는 그 요소를 그린다. 그림은 프로그람에서 그 입구에 결합할수 있다. 즉 탕크에 발브, 관에 탕크, …를 결합할수 있다. 사용자는 탕크가 찼는가, 비였는가, 발브가 열렸는가, 닫겼는가를 감시할수 있다. 관열에 있는 수들은 그 내부에서 물의 흐름을 표시한다. 이것은 체계의 조작을 개발하기 쉽게 한다. 또한 여기서 수행하지 않는것까지 실현하면 프로그람은 더 커지고 복잡해진다.

요 약

제작자가 제공하는 객체서고는 보통 .H머리부파일안에 클라스서고를 포함하는 공개요소(대면부)와 .OBJ파일이나 .LIB서고파일안에 성원함수정의를 포함하는 비공개요소로서 배포된다.

C++번역프로그람은 여러개의 원천과 목적파일들을 하나의 실행가능파일에 결합한다. 이것은 한 제작자가 제공한 파일들과 다른 제작자가 제공한 파일들을 결합하여 최종적인 응용프로그람을 창조하게 한다. 프로젝트의 특성은 어떤 파일을 번역하여야 하는가 하는 리력을 유지하는것이다. 마지막 변경후 변경된 원천파일을 번역하고 결과로 생기는 목적파일들을 련결한다.

프로젝트

1. VeryLong실례에서 VeryLong클라스를 위한 덜기와 나누기를 하는 성원함수들을 작성하시오. 이것은 -와 /연산자를 재정의해야 한다. 여기에는 몇가지 문제가 있다. 덜기를 포함할 때에는 VeryLong이 정수는 물론 부수일 때에도 동작해야 한다. 이것은 더하기와 곱하기루틴을 더 복잡하게 한다.

나누기를 처리하기 위하여 긴 수의 나누기를 설계하시오. 그다음 이 단계를 나누기성원함수로 쓰시오. 비교가 필요하므로 비교루틴을 쓸 필요가 있다.

2. Pipes실례에 Pump클라스를 추가하시오. 따라서 중력에 관계되지 않는 급수체계를 모형화할수 있다. 한개 클라스처럼 포함하는 급수체계를 만드시오.(암시: Pump는 Tank로부터 그 일부를 파생시켜야 한다.)

3. 자기가 흥미를 가지는 어떤 객체를 모의하는 클라스서고를 만드시오. 그것을 시험하는 main()의뢰기프로그람을 쓰시오. 두개의 클라스서고를 더 풍부하게 하시오.

제 14 장. 형판과 례외

이 장에서는 C++의 고급한 두가지 특성 즉 형판과 례외를 소개한다. 형판(template)은 한개의 함수 또는 클라스를 사용하여 각이한 많은 자료형을 조종하게 한다. 례외(exception)는 클라스안에서 발생하는 오유를 조종하는 일관적이고 유일한 방법을 제공한다.

형판개념은 두가지 방법으로 즉 함수 및 클라스와 함께 사용할수 있다.

이 장에서는 우선 함수형판과 클라스형판에 대하여 설명하고 례외와 다중례외, 인수있는 례외, 내부례외에 대하여 설명한다.

제 1 절. 함수형판

어떤 수의 절대값을 돌려주는 함수를 쓰려고 한다. 수의 절대값은 그 수의 부호에 관계없는 값이다. 3의 절대값은 3, -3의 절대값은 3이다.

보통 이 함수는 자료형별로 정의한다.

```
int Abs(int n)
{
   return (n < 0) ? -n : n;
}
```

여기서 함수는 int형의 인수를 가지고 같은 형의 값을 돌려주는것으로 정의된다. 그러나 long형의 절대값을 구하려고 한다고 하자. 이때 새로운 함수를 다시 써야 한다.

```
long Abs(long n)
{
   return (n < 0) ? -n : n;
}
```

float형에 대해서는

```
float Abs(float n)
{
   return (n < 0) ? -n : n;
}
```

함수본체는 매번 같은 방법으로 쓰지만 함수들은 다른 형의 인수와 돌림값을 가지는 전혀 다른 함수이다. C++에서는 이 함수들을 모두 같은 이름으로 재정의할수 있지만 매개 함수에 대하여 따로따로 정의를 써야 한다.(C언어에서는 재정의를 유지하지 않으므로 다른 형의 함수는 같은 이름을 가질수 없다. C함수서고에는 abs(), fabs(), fabc1(), labs(), cabs()와 같은 류사한 함수무리가 있다.)

같은 함수이름을 다른 형에 대하여 반복하여 쓰는것은 시간을 소비하고 코드에서 공간을 차지한다. 또한 그러한 함수 하나에서 오유를 발견하면 매개 함수본체를 모두

수정해야 한다.

같은 처리를 하는 함수를 한번만 쓰는 방법으로 다른 자료형에 대해서도 동작하게 하면 좋다. 함수형판이 이것을 수행한다. 그 원리를 그림 14-1에서 계층적으로 보여준다.

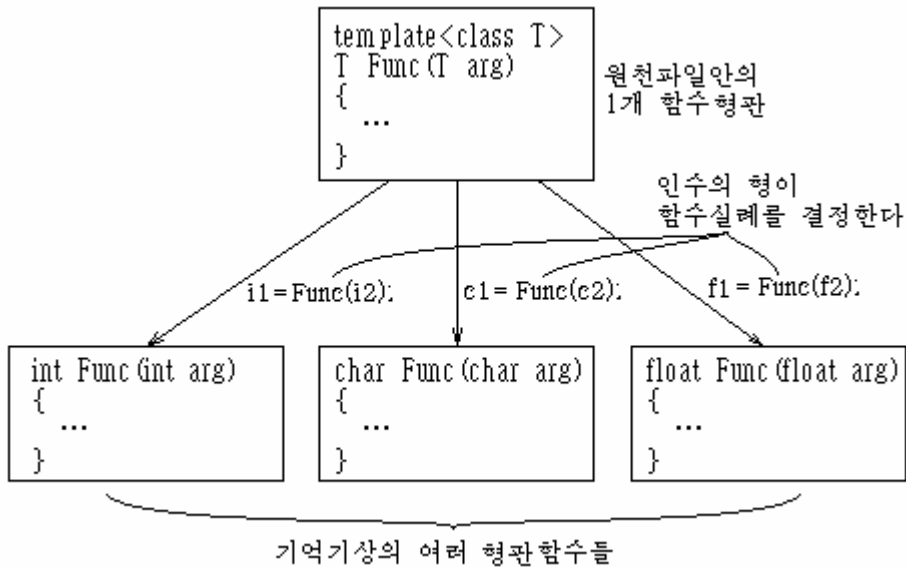



그림 14-1. 함수형판.

1. 간단한 함수형판

첫 실례는 절대값함수를 형판로 만드는 방법을 보여준다. 이 함수형판은 기본수값형에 대하여 작업한다. 프로그람은 Abs()의 형판을 정의하고 각이한 자료형에 대하여 이 함수를 호출하여 그 동작을 판정한다.

(실례 14-1) 절대값함수형판

```
#include <iostream>
using namespace std;
template <class T>
T Abs(T n)
{
   return (n < 0) ? -n : n;
}
int main()
{
   int int1 = 5;
   int int2 = -6;
   long lon1 = 70000L;
   long lon2 = -80000L;
   double dub1 = 9.95;
   double dub2 = 10.15;
```

```
    cout << "\nAbs(" << int1 << ")=" << Abs(int1);
    cout << "\nAbs(" << int2 << ")=" << Abs(int2);
    cout << "\nAbs(" << lon1 << ")=" << Abs(lon1);
    cout << "\nAbs(" << lon2 << ")=" << Abs(lon2);
    cout << "\nAbs(" << dub1 << ")=" << Abs(dub1);
    cout << "\nAbs(" << dub2 << ")=" << Abs(dub2);
    cout << endl;
    return 0;
}
```

프로그램의 출력은 다음과 같다.

```
Abs(5)=5
Abs(-6)=6
Abs(70000)=70000
Abs(-80000)=80000
Abs(9.95)=9.95
Abs(-10.15)=10.15
```

Abs()함수는 인수로 사용하는 세개의 자료형(int, long, double)에 대하여 모두 동작한다. 다른 기본자료형에 대해서도 물론 동작하고 작기연산자(<)와 단항미누스연산자(-)를 적당히 재정의하면 사용자정의자료형에 대해서도 동작한다.

여기에 Abs()함수가 여러 자료형과 작업하게 하는 방법이 있다.

```
template <class T>
T Abs(T n)
{
    return (n < 0) ? -n : n;
}
```

이 문법은 첫 행에서 예약어 template로 시작하고 다음 행에 함수정의가 온다. 이것을 함수형판(function template)이라고 부른다. Abs()를 정의하는 새로운 방법이 어떻게 그와 같은 융통성을 주는가?

1) 함수형판문법

함수형판에서 기본혁신은 int와 같은 특정한 형의 함수들에 의해서가 아니라 임의의 형을 가질수 있는 이름에 의해 사용되는 자료형을 표시하는것이다. 앞의 함수형판에서 그 이름은 T이다. (이것은 작성자의 요구대로 쓸수 있다. 즉 Type, AnyType, 또는 FacBar 등)

template예약어는 함수형판을 정의하려고 한다고 번역프로그람에 신호한다. <>안의 예약어 class는 형을 의미한다. 앞에서 본것처럼 클라스를 사용하는 자료형을 정의할수 있으므로 형과 클라스사이의 구별은 사실상 없다. 예약어 class뒤에 오는 변수(실례로 T)를 형판인수(template argument)라고 한다.

함수정의에서 int와 같은 특정자료형을 쓸수 있는것은 항상 형판인수 T로 대신할수 있다. Abs()함수에서는 그 이름이 두번 즉 첫 행(함수선언)의 인수형과 돌림값의 형으로서 나타난다. 복잡한 함수들에서는 함수본체에서도 변수를 정의할 때 여러번 나타

날수 있다.

2) 번역프로그람의 동작

template예약어가 나타나고 그 뒤에 함수정의가 올 때 번역프로그람은 무엇을 하는가? 물론 다른것은 없다. 함수형판자체는 번역프로그람이 어떤 코드도 생성하게 하지 못한다. 그것은 아직 함수와 작업하는 자료형을 모르기때문이다. 단순히 앞으로의 가능한 사용을 위하여 형판을 선언한다.

코드는 프로그람의 명령문에 의하여 실제로 호출될 때까지 생성되지 않는다. 실례 14-1에서 이것은 명령문

```
cout << "\nAbs(" << int1 << ")=" << Abs(int1);
```

의 Abs(int1)과 같은 식에서 나타난다.

번역프로그람은 이러한 함수호출을 발견하면 인수 int1의 형이 int이므로 사용하려는 형이 int라는것을 알게 된다. 따라서 함수형판에서 이름 T가 있는 곳을 int로 바꿈으로써 int형의 Abs()전용판을 생성한다.

이것을 함수형판의 실례화(instantiation)라고 한다. 매개의 실례화된 함수를 형판함수(template function)라고 한다.(즉 형판함수는 함수형판의 특정한 실례이다.)

또한 번역프로그람은 새로운 실례화된 함수에로의 호출을 생성하고 Abs(intl)이 있는 코드에 그것을 삽입한다. 이와 류사하게 식 Abs(lonl)은 번역프로그람이 long형에 대하여 조작하는 Abs()판을 생성하고 이 함수에로의 호출을 생성한다. 또한 Abs(dubl)은 double에 대해 작업하는 함수를 생성한다. 물론 번역프로그람은 매개 자료형에 대하여 Abs()의 한개 판만 생성하면 충분하다. 이처럼 int판의 함수호출이 두개 있어도 그 판의 코드는 실행가능판에 한번만 나타난다.

3) 목록의 단순화

프로그람에 사용하는 RAM의 량은 형판수법을 사용하든 세개의 서로 다른 함수를 쓰든 관계없이 같다. 절약되는것은 원천파일에 세개의 함수들중 하나만 입력하는것이다. 이것은 프로그람을 더 짧고 리해하기 쉽게 만든다. 또한 함수가 작업하는 방법을 변경하려면 세곳대신에 한곳에서만 변경하면 된다.

4) 인수

번역프로그람은 함수호출에서 인수로 쓰이는 전체 자료형에 기초하여 함수번역방법을 결정한다. 함수의 돌림값형은 인수에 의존하지 않는다. 이것은 여러개의 재정의된 함수들중 어느것을 호출하는가를 번역프로그람이 결정하는 방법과 비슷하다.

5) 다른 종류의 설계도

함수형판은 기억기에 실제로 프로그람코드를 배치하지 못하므로 실제적인 함수가 아니다. 그대신에 함수형판은 많은 함수를 만들기 위한 견본이다. 이것은 클라스를 구체화하지 않는 방법과 비슷하지만 류사한 많은 객체들을 만들기 위한 설계도와 같다.

2. 여러 인수를 가지는 함수형판

함수형판의 다른 실례를 고찰해보자. 이 실례는 세개의 인수를 가진다. 즉 두개는 형판인수, 다른 하나는 기본형의 인수이다. 함수는 배렬에서 특정값을 검색하여 발견하면 그 값의 배렬원소를 돌려주고 찾지 못하면 -1을 돌려준다. 인수로서는 배렬에로의 지적자, 검색값, 배렬의 크기이다. main()에서는 서로 다른 형의 배렬을 4개 정의하고 4개 값을 검색한다. 그다음 매개 배렬에 대하여 형판함수를 한번씩 호출한다. 여기에 실례 14-2의 프로그람이 있다.

(실례 14-2) 배렬의 계수기를 찾는 함수에 사용하는 형판

```
#include <iostream>
using namespace std;
template <class AType>
int Find(AType array[], AType value, int size)
{
  for(int j=0; j<size; j++)
     if(array[j] == value)
         return j;
  return -1;
}
char chrArr[] = { 1, 3, 5, 9, 11, 13 };
char ch = 5;
int intArr[] = { 1, 3, 5, 9, 11, 13 };
int in = 6;
long lonArr[] = { 1L, 3L, 5L, 9L, 11L, 13L };
long lo = 11L;
double dubArr[] = { 1.0, 3.0, 5.0, 9.0, 11.0, 13.0 };
double db = 4.0;
int main()
{
  cout << "\nchrArray에서 수 5의 첨수=" << Find(chrArr, ch, 6);
  cout << "\nintArray에서 수 6의 첨수=" << Find(intArr, in, 6);
  cout << "\nlonArray에서 수 11의 첨수=" << Find(lonArr, lo, 6);
  cout << "\ndouArray에서 수 4의 첨수=" << Find(dubArr, db, 6);
  cout << endl;
  return 0;
}
```

여기서 형판인수는 AType이고 함수인수는 세개이다. 첫째 인수는 배렬에로의 지적자, 둘째 인수는 대조하려는 항목의 형이며 셋째 인수 즉 배렬크기는 항상 int형으로서 형판인수가 아니다. 프로그람의 출력은 다음과 같다.

```
chrArray에서 수 5의 첨수=2
intArray에서 수 6의 첨수=-1
lonArray에서 수 11의 첨수=4
douArray에서 수 4의 첨수=-1
```

번역프로그람은 4개의 각이한 판의 함수를 생성하고 그중 하나를 그 호출에 사용

한다. 문자배렬의 첨수 2에서 5를 찾고 옹근수배렬에서는 첨수 4에서 11을 찾는다.

1) 형판인수들은 일치해야 한다

형판함수를 호출할 때 같은 형판인수의 모든 실례들은 같은 형이 되여야 한다. 실례로 Find()에서 배렬이름은 int형, 탐색하려는 값도 int형으로 되여야 한다.

따라서 다음과 같이 쓸수 없다.

```
int intArray[] = {1,2,5,7};
float f1 = 5.0;
int f1 = find(intArray, f1, 4);
```

그것은 번역프로그람이 AType의 모든 실례가 같은 형일것을 요구하기때문이다. 번역프로그람은 함수 Find(int*, int, int);를 생성하지만 Find(int*, float, int);를 생성할수 없다. 그것은 첫째와 둘째 인수의 형이 같아야 하기때문이다.

2) 문법변경

일부 프로그람작성자들은 template예약어를 쓰고 같은 행에 함수선언자를 쓴다.

```
template <class AType> int Find(AType array*, AType value, int size)
{
  // 함수본체
}
```

물론 번역프로그람은 이 형태도 정확히 번역하지만 여러행에 나누어 쓰는 수법보다 리해하기 힘들다.

3) 한개이상의 형판인수

매개 함수형판에 한개이상의 형판인수를 사용할수 있다. 실례로 Find()함수형판와는 달리 적용하려는 배렬의 크기를 확인할수 없다고 하자. 배렬이 너무 크면 배렬크기에 int형대신 long형이 요구된다. 다른 한편 그럴 필요가 없을 때에는 long형을 사용하지 않는다. 함수를 호출할 때 보관하는 자료의 형은 물론 배렬크기의 형을 선택하려고 한다. 그러기 위하여 배렬크기를 형판인수로 할수 있다. 그것을 BType라고 하자.

```
template <class AType, class BType>
int Find(AType array*, AType value, BType size)
{
   for(BType j=0; j<size; j++)
      if(array[j] == value)
         return j;
   return static_cast<Btype>(-1);
}
```

크기에 int형이나 long형(지어 사용자정의자료형)을 사용할수 있다. 번역프로그람은 배렬형과 검색값뿐아니라 배렬크기의 형에 기초하여 서로 다른 함수들을 생성한다.

서로 다른 형판인수들은 형판로부터 많은 함수의 실례를 만들게 한다. 배렬이 사용할수 있는 기본형이 6가지 있으면 두개의 형판인수는 36개의 함수를 창조할수 있다. 이것은 함수가 클 때 많은 기억기를 소비하게 하므로 그 함수를 실제로 호출하지 않는다면 그것을 실례화하지 않는다.

4) 마크로가 왜 없는가?

C프로그람작성자들은 각이한 자료형에 대하여 각이한 판의 함수를 창조할 때 마크로를 사용한다. 실례로 Abs()함수는 다음과 같이 정의할수 있다.

```
#define Abs(n) ((n<0)?(-n):n))
```

이것은 마크로가 단순한 본문교체를 수행하고 임의의 형에 대하여 동작하므로 실례 14-1의 클라스형판과 류사한 효과를 가진다. 그러나 이미 언급한것처럼 대체로 마크로는 C++에서 사용하지 않는다. 그와 관련한 몇가지 문제가 있다. 하나는 마크로가 형검사를 하지 못하는것이다. 같은 형이여야 하는 인수가 마크로에 여러개 있을수 있으나 번역프로그람은 그 형을 검사하지 않는다. 또한 돌림값의 형이 정의되지 않으므로 번역프로그람은 적합하지 않은 변수에 그것을 대입하는지 알수 없다. 어떤 경우에 마크로는 단일명령문으로 표시되는 함수로 극한된다. 마크로과 관련한 다른 문제도 있다. 총체적으로 마크로는 피하는것이 좋다.

5) 무엇이 작업하는가?

부정자료형에 대하여 형판함수를 실례화할수 있는가를 어떻게 아는가?

실례 14-2로부터 C문자들의 배렬로 된 C문자렬(char*형)을 검색할 때 Find()를 사용할수 있는가?

그 가능성을 알아내기 위하여 함수에서 사용하는 연산자를 검사한다. 연산자들이 그 자료형에 대하여 모두 동작하면 그것을 적당히 사용할수 있다. 그러나 Find()에서는 같기(==)연산자를 사용하여 두 변수를 비교한다. C문자렬에 이 연산자를 사용할수 없고 strcmp()서고함수를 사용해야 한다.

따라서 Find()는 C문자렬에서 동작하지 않는다. 그러나 string클라스에서 ==연산자를 재정의하면 그대로 동작한다.

6) 일반함수로 시작한다

형판함수를 쓸 때 int라는 다른 기본형에 대해 동작하는 일반함수로 시작하는것이 더 좋다. 그러면 형판인수와 여러 형에 대하여 걱정하지 않고 설계하며 오유수정도 할수 있다. 그다음 모든것이 제대로 작업할 때 함수정의를 형판로 바꾸고 다른 형에 대하여 작업하는가 검사한다.

제 2 절. 클라스형판

형판개념을 클라스에로 확장할수 있다. 일반적으로 클라스형판(class template)는 자료보관(용기)클라스에 사용된다. 이미 앞에서 취급한 탄창과 련결목록이 자료보관클라스의 례이다. 그러나 우리가 제공하는 클라스의 실례들은 단순히 기본형의 자료만 보관한다. 실례 7-8에서 Stack클라스는 int형의 자료만 보관한다. 여기에 그 클라스가 있다.

```
class Stack
{
private:
   int st[MAX];
   int top;
public:
   Stack();
   void Push(int var);
   int Pop();
};
```

탄창에 long형의 자료를 보관하려면 완전히 새로운 클라스를 정의해야 한다.

```
class LongStack
{
private:
   long st[MAX];
   int top;
public:
   Stack();
   void Push(long var);
   long Pop();
};
```

마찬가지로 우리가 보관하려고 하는 매개의 자료형에 대하여 새로운 탄창클라스를 창조하여야 한다. 단순한 기본형대신에 모든 형의 변수들에 대하여 작업하는 단일한 클라스명세를 쓸수 있으면 좋다. 클라스형판이 이 기능을 수행한다. 클라스형판을 사용하는 실례 14-3이 있다.

(실례 14-3) 탄창클라스형판

```
#include <iostream>
using namespace std;
const int MAX = 100;
template <class Type>
class Stack
{
private:
   Type st[MAX];
   int top;
public:
   Stack() { top = -1; }
   void Push(Type var) { st[++top] = var; }
   Type Pop() { return st[top--]; }
};
template<float> Stack;
int main()
{
   Stack<float> s1;
   s1.Push(1111.1F);
   s1.Push(2222.2F);
```

```
    s1.Push(3333.3F);
    cout << "1: " << s1.Pop() << endl;
    cout << "2: " << s1.Pop() << endl;
    cout << "3: " << s1.Pop() << endl;
    Stack<long> s2;
    s2.Push(1231231);
    s2.Push(2342342);
    s2.Push(3453453);
    cout << "1: " << s2.Pop() << endl;
    cout << "2: " << s2.Pop() << endl;
    cout << "3: " << s2.Pop() << endl;
    return 0;
}
```

여기서 클라스 Stack는 형판클라스로서 제공된다. 그 수법은 함수형판과 같다. template예약어와 class Stack는 이 클라스가 형판라는것을 알린다.

```
template <class Type>
class Stack
{
  // 자료와 형판인수를 사용하는 함수들
};
```

실례에서 Type라는 형판인수는 배렬 st형에로의 참고가 있는 클라스명세의 모든 곳(int와 같은 고정자료형객체)에서 사용된다. 그러한 위치는 세곳(즉 st의 정의, Push()함수의 인수형, Pop()함수의 돌림값형)이 있다.

클라스형판의 실례화방법은 함수형판과 다르다. 함수형판로부터 실례화된 함수를 창조하려면 특정한 형의 인수를 사용하여 함수를 호출한다. 그러나 클라스는 형판인수를 사용하여 객체를 정의할 때 실례화된다.

```
Stack<float> s1;
```

이것은 float형의 수값을 보관하는 탄창인 객체 s1을 창조한다. 번역프로그람은 클라스명세에서 형판인수 Type가 나타날 때 항상 float형을 사용하여 이 객체의 자료를 위한 공간을 기억기에 제공한다. 또한 Stack(float)형의 다른 객체들에 의하여 기억기안에 보관되지 않았으면 성원함수용 공간도 제공한다. 또한 성원함수들은 float형에 대하여 제대로 동작한다. 그림 14-2는 클라스형판과 특정한 객체의 정의에 의하여 기억기에 어떻게 보관되는가를 보여준다.

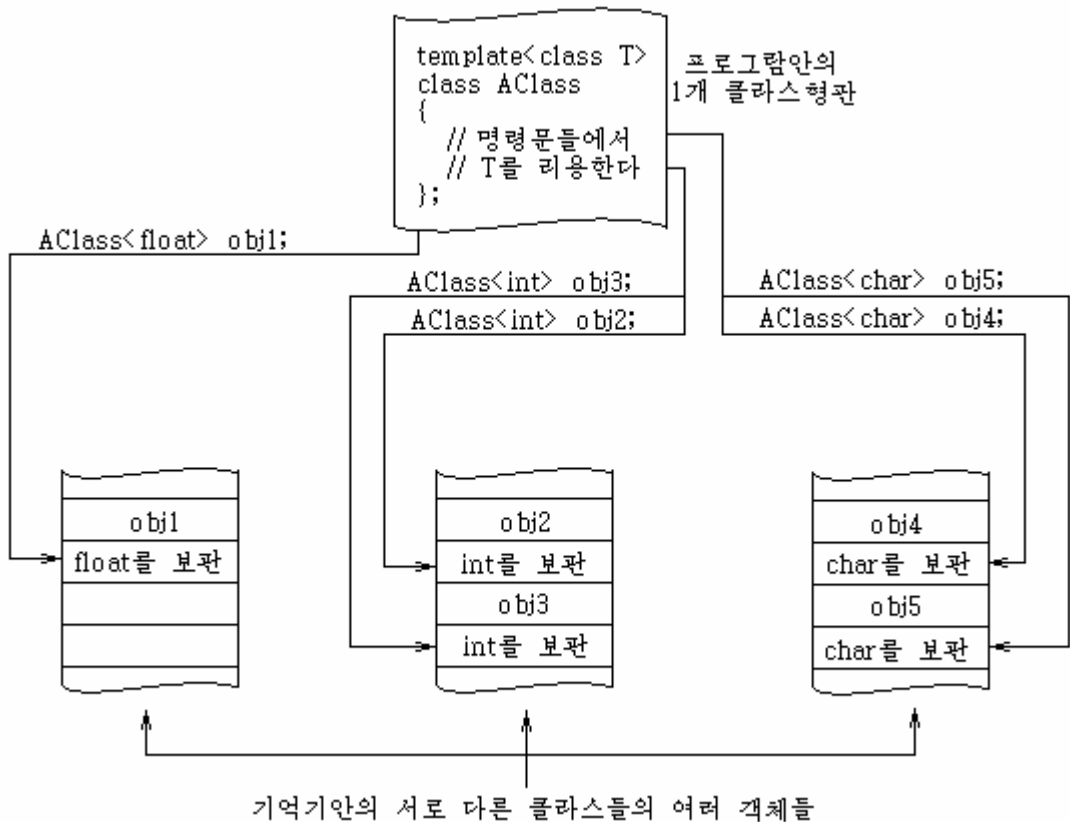



그림 14-2. 클라스형판

각이한 형의 객체를 보관하는 stack객체를 창조할 때 실례로 Stack<long> s2;은 자료용의 다른 공간뿐아니라 long형에 대하여 조작하는 성원함수들의 새로운 일식을 창조한다.

s1의 형이름은 클라스이름 Stack와 형판인수로 이루어진다. 즉 Stack(float)이다. 이것은 Stack(int) 또는 Stack(long)과 같이 형판로부터 창조할수 있는 다른 클라스로부터 그것을 구별한다. 실례 14-3에서는 s1과 s2탄창들에 값을 밀어넣고 꺼내며 그때마다 꺼낸 값을 표시한다.

출력은 다음과 같다.

```
1: 3333.3
2: 2222.2
3: 1111.1
1: 3453453
2: 2342342
3: 1231231
```

이 실례에서 형판수법은 하나의 가치를 가지는 두개의 클라스를 제공하고 다른 수값형에 대하여 한행의 코드를 사용하여 클라스객체들을 실례화할수 있다.

1. 클라스이름은 상황에 의존한다

실례 14-3에서 클라스형판의 성원함수들은 모두 클라스안에 정의된다. 만일 성원함수가 클라스명세의 밖에서 정의되면 새로운 문법이 요구된다. 다음의 프로그람은 이것을 보여준다.

(실례 14-4) 탄창클라스형판의 성원함수를 형판밖에서 정의

```
#include <iostream>
using namespace std;
const int MAX = 100;
template <class Type>
class Stack
{
private:
   Type st[MAX];
   int top;
public:
   Stack();
   void Push(Type var);
   Type Pop();
};
template <class Type>
Stack<Type>::Stack()
{
   top = -1;
}
template <class Type>
void Stack<Type>::Push(Type var)
{
   st[++top] = var;
}
template <class Type>
Type Stack<Type>::Pop()
{
   return st[top--];
}
int main()
{
   Stack<float> s1;
   s1.Push(1111.1F);
   s1.Push(2222.2F);
   s1.Push(3333.3F);
   cout << "1: " << s1.Pop() << endl;
   cout << "2: " << s1.Pop() << endl;
   cout << "3: " << s1.Pop() << endl;
   Stack<long> s2;
   s2.Push(1231231);
   s2.Push(2342342);
```

```
    s2.Push(3453453);
    cout << "1: " << s2.Pop() << endl;
    cout << "2: " << s2.Pop() << endl;
    cout << "3: " << s2.Pop() << endl;
    return 0;
}
```

식 templat<class Type>는 클라스정의뿐아니라 외부에 정의된 성원함수의 앞에 있어야 한다. 여기에 Push()함수가 있다.

```
template <class Type>
void Stack<Type>::Push(Type var)
 {
    st[++top] = var;
}
```

이름 Stack<Type>는 Push()가 성원인 클라스를 식별하는데 쓰인다. 보통의 비형판성원함수에서는 이름 Stack만 앞에 놓을수 있다.

```
void Stack::Push(int var)
 {
    st[++top] = var;
}
```

그러나 함수형판에 대해서는 형판인수 즉 Stack<Type>를 요구한다.

이처럼 형판클라스의 이름은 각이한 상황에서 서로 다르게 표시된다. 클라스선언에서 이것은 단순히 클라스이름 그 자체 즉 Stack이다. 외부에 정의된 성원함수에서는 클라스이름과 형판인수이름 즉 Stack<Type>이다. 특정한 자료형을 보관하기 위한 실례객체를 정의할 때에는 클라스이름과 그 특정형 즉 Stack<float>이다.

```
class Stack                                   // 탄창클라스선언지정자
    {};
void Stack<Type>::Push(Type var)     // Push()정의
    {}
Stack<float> s1;                              // Stack<float>형객체의 정의
```

경우에 따라서 정확한 이름을 사용하여야 한다. Stack에 <Type>대신 <float>를 추가하는것을 잊기 쉽다.

실례에서는 그것을 보여주지 않지만 성원함수가 그 자체클라스의 값을 돌려줄 때 문법을 주의해야 한다. 8장 련습 4에서 취급한것처럼 옹근수에 안전성특성이 제공된 클라스 Int를 정의한다고 가정하자. Int형을 돌려주는 성원함수 XFunc()에 외부정의를 사용한다면 범위해결연산자를 앞에 놓는것은 물론이고 돌림값에 Int<Type>를 써야 한다.

```
Int<Type> Int<Type>::XFunc(Int arg)
    {}
```

다른 한편 함수인수의 행에서 사용된 클라스이름은 <Type>지정을 포함하지 말아야 한다.

2. 형판을 사용하는 련결목록클라스

형판을 자료보관클라스에 사용하는 다른 실례를 고찰하자. 이것은 실례 10-23의 변경판이다. 여기서는 LinkList클라스자체를 형판로 만든다. 실례로 매개 자료항목을 보관하는 Link구조체를 형판로 만든다.

(실례 14-5) 형판로 실현한 련결목록

```
#include <iostream>
using namespace std;
template <class Type>
struct Link
{
  Type data;
  Link* next;
};
template <class Type>
class LinkList
{
private:
  Link<Type>* first;
public:
  LinkList() { first = NULL; }
  void AddItem(Type d);
  void Display();
};
template <class Type>
void LinkList<Type>::AddItem(Type d)
{
  Link<Type>* newLink = new Link<Type>;
  newLink->data = d;
  newLink->next = first;
  first = newLink;
}
template <class Type>
void LinkList<Type>::Display()
{
  Link<Type>* current = first;
  while(current != NULL)
  {
     cout << endl << current->data;
     current = current->next;
  }
}
int main()
{
  LinkList<double> ld;
  ld.AddItem(151.5);
  ld.AddItem(262.6);
```

```
    ld.AddItem(373.7);
    ld.Display();
    LinkList<char> lch;
    lch.AddItem('a');
    lch.AddItem('b');
    lch.AddItem('c');
    lch.Display();
    cout << endl;
    return 0;
}
```

main()에서는 두개의 련결목록을 정의한다. 그중 하나는 double형의 수를 보관하고 다른 하나는 char형의 문자를 보관한다. 그다음 AddItem()성원함수로 매개 련결목록에 세개 항목씩 보관하고 Display()성원함수로 그 항목들을 모두 표시한다. 실례 14-5의 출력은 다음과 같다.

```
373.7
262.6
151.5
c
b
a
```

LinkList클라스와 Link구조체는 둘다 임의의 형에 대하여 성립하는 형판인수 Type를 사용한다.

따라서 LinkList뿐아니라 Link도 형판이고 다음 행이 앞에 놓인다.

```
template<calss Type>
```

Link는 형판로 전환된 클라스가 아니다. 일반자료형을 사용하는 구조체는 Link와 같이 형판로 전환되여야 한다.

앞에서와 같이 클라스가 프로그람의 각이한 부분에서 어떻게 이름지어지는가에 대하여 주목해야 한다. 자기의 선언안에서 클라스 또는 구조체 자체의 이름 즉 LinkList와 Link를 사용할수 있다. 외부성원함수들에서는 클라스 혹은 구조체이름과 형판인수 즉 LinkList<Type>를 사용해야 한다. 실제로 LinkList형의 객체를 정의할 때 목록에 보관하려는 고유한 자료형을 사용해야 한다. 즉

```
LinkList<double> ld;
```

3. 사용자정의자료형의 보관

지금까지 프로그람에서는 형판클라스들을 기본자료형을 보관하는데만 사용하였다. 실례 14-5에서는 련결목록에 double과 char형의 수들을 보관하였다. 이 형판클라스들에서 사용자정의자료형(클라스)을 보관할수 있는가? 물론 보관할수 있다.

1) 종업원목록

실례 9-5의 Employee클라스를 시험하자.

실례 14-5의 련결목록에 Employee형의 객체들을 보관할수 있는가?

형판함수에서처럼 형판클라스가 특정한 클라스의 객체들에 대하여 수행하는 조작을 검사함으로써 그 클라스의 객체들이 형판클라스에서 동작하는가 알아볼수 있다. LinkList클라스는 재정의된 삽입(<<)연산자에 의하여 거기에 보관된 객체들을 표시한다.

```
void LinkList<Type>::Display()
{
  ...
  cout << endl << current->data;
}
```

이것은 기본형과 관련한 문제가 아니다. 즉 삽입연산자는 이미 정의되여있다. 그러나 실례 9-5의 Employee클라스는 재정의된 삽입연산자를 가지지 않으므로 이 연산자를 포함하도록 Employee를 변경해야 한다. 또한 사용자로부터 종업원의 자료를 간단히 얻기 위하여 발취(>>)연산자도 재정의한다. 발취연산자에 의해 자료는 련결목록에 추가되기 전에 림시객체 empTemp에 배치된다.

(실례 14-6) 형판로 실현한 련결목록

```
#include <iostream>
using namespace std;
const int LEN = 80;
class Employee
{
private:
  char name[LEN];
  unsigned long number;
public:
  void GetData()
  {
     cout << "\n  이름? "; cin >> name;
     cout << "\n  종업원 번호? "; cin >> number;
  }
  void PutData() const
  {
     cout << "\n  이름: " << name;
     cout << "\n  종업원 번호: " << number;
  }
  friend istream& operator>> (istream& s, Employee& e);
  friend ostream& operator<< (ostream& s, Employee& e);
};
istream& operator>> (istream& s, Employee& e)
{
  cout << "\n  이름? "; cin >> e.name;
  cout << "\n  종업원 번호? "; cin >> e.number;
  return s;
}
ostream& operator<< (ostream& s, Employee& e)
```

```
{
   cout << "\n  이름: " << e.name;
   cout << "\n  종업원 번호: " << e.number;
   return s;
}
template <class Type>
struct Link
{
   Type data;
   Link* next;
};
template <class Type>
class LinkList
{
private:
   Link<Type>* first;
public:
   LinkList() { first = NULL; }
   void AddItem(Type d);
   void Display();
};
template <class Type> void LinkList<Type>::AddItem(Type d)
{
   Link<Type>* newLink = new Link<Type>;
   newLink->data = d; newLink->next = first; first = newLink;
}
template <class Type> void LinkList<Type>::Display()
{
   Link<Type>* current = first;
   while(current != NULL)
   {
      cout << endl << current->data;
      current = current->next;
   }
}
int main()
{
   LinkList<Employee> lEmp;
   Employee empTemp;
   char ans;
   do
   {
      cin >> empTemp;
      lEmp.AddItem(empTemp);
      cout << "\n계속하겠습니까(y/n)?"; cin >> ans;
   } while(ans != 'n');
   lEmp.Display();
   cout << endl;
```

```
    return 0;
}
```

main()에서는 lEmp라는 련결목록의 실례를 만든다. 그다음 순환에서 사용자에게 종업원자료를 입력하게 하고 목록에 그 종업원객체를 추가한다. 사용자가 순환을 완료할 때 종업원자료를 모두 표시한다. 여기에 프로그람의 대화가 있다.

```
  이름? 김인철
  종업원 번호? 1233
계속하겠습니까(y/n)? y
  이름? 리영남
  종업원 번호? 2344
계속하겠습니까(y/n)? y
  이름? 최철호
  종업원 번호? 3455
계속하겠습니까(y/n)? n
  이름 = 최철호
  종업원 번호 = 3455
  이름 = 리영남
  종업원 번호 = 2344
  이름 = 김인철
  종업원 번호 = 1233
```

Employee형의 객체를 보존하기 위해 LinkList클라스를 조금도 수정할 필요는 없다. 이것은 형판클라스의 우점이다. 형판클라스는 기본형뿐아니라 사용자정의자료형과도 잘 작업한다.

2) 무엇을 보관할수 있는가?

자료보관형판클라스의 연산자함수들을 검사하여 형판클라스에 특정한 형의 변수들을 보관할수 있는가를 알아볼수 있다.

실례 14-6에서 LinkList클라스안에 문자렬(string클라스)을 보관할수 있는가? string클라스의 성원함수들은 삽입(<<)과 발취(>>)연산자이다. 이 연산자들은 문자렬과 아주 잘 작업하므로 문자렬보관에 LinkList클라스를 사용못할 근거는 없다. 그러나 특정자료형에 대해 동작하지 않는 성원함수가 보관클라스에 존재하면 그 자료형의 보관에 형판클라스를 사용할수 없다.

제 3 절. 례외

례외는 C++클라스들에 의하여 발생하는 실행시오유를 조종하기 위한 계통적인 객체지향수법을 제공한다. 례외는 실행시에 발생하는 오유이다. 례외는 기억기를 벗어나는 실행, 파일을 열수 없는것, 객체를 불가능한 값으로 초기화하려는것, 벡토르의 첨수경계밖에서의 사용과 같은 광범히 변화하는 례외적인 환경에 의해 발생한다.

1. 례외의 필요성

그러면 오유를 조종하기 위한 새로운 기구가 왜 필요한가?

이전에 처리가 어떻게 조종되였는가를 고찰하자.

C언어프로그람에서 오유는 자주 그것이 발생하는 함수로부터 특정값을 돌려주는 방법으로 경고하였다. 실례로 디스크파일함수는 NULL 또는 0을 보내여 오유를 경고하므로 이 함수들을 호출할 때마다 돌림값을 검사해야 한다.

```
if(someFunc() == ERROR_RETURN_VALUE)
    // 오유조종 혹은 오유처리함수호출
else
    // 표준처리
if(anotherFunc() == NULL)
    // 오유조종 혹은 오유처리함수호출
else
    // 표준처리
if(thirdFunc() == 0)
    // 오유조종 혹은 오유처리함수호출
else
    // 표준처리
```

여기서 하나의 문제는 함수를 호출할 때마다 프로그람에서 검사하여야 하는것이다. 매번의 함수호출은 if~else명령문안에 놓이고 오유조종명령문(또는 오유처리함수호출)을 추가하는것은 많은 코드를 요구하므로 프로그람을 복잡하고 읽기 힘들게 만든다.

문제는 클라스를 사용할 때 더 복잡해지는것이다. 즉 명시적으로 호출되는 함수가 없이 오유가 발생한다. 실례로 어떤 클라스의 객체를 정의한다고 가정하자. 즉

```
SomeClass obj1, obj2, obj3;
```

클라스구성자에서 오유가 발생하면 그것을 어떻게 찾겠는가?

구성자는 암시적으로 호출되므로 검사할 돌림값이 없다.

이것은 응용프로그람이 클라스서고를 사용할 때 더 복잡해진다. 클라스서고를 자주 사용하는 응용프로그람은 개별적사람들이 작성한다. 즉 클라스서고는 제작자가 작성하고 응용프로그람은 클라스서고를 구입한 작성자가 작성한다. 이것은 클라스성원함수로부터 함수를 호출하는 프로그람까지 련관되도록 오유값들을 배렬하기 힘들게 한다. 클라스선언안의 깊은 곳에서 오유가 발생하는 문제는 례외에 의하여 해결할수 있는 가장 중요한 문제이다. C프로그람작성자들은 오유를 포착하는 다른 방법 즉 setjmp()와 longjmp()함수결합을 알고있다. 그러나 이 수법은 객체의 해체를 적당히 조종하지 못하므로 객체지향환경에서 적합하지 않다.

2. 례외문법

일정한 클라스의 객체들을 창조하고 그것과 교제하는 응용프로그람을 고찰하자.

보통 클라스성원함수에 대한 응용프로그람의 호출은 문제없다. 그러나 때때로 응

용프로그람은 성원함수안에서 오유를 발생시킨다.

그때 성원함수는 오유가 발생하였다는것을 응용프로그람에 알린다. 례외를 사용할 때 성원함수에서 오유가 발생하는것을 례외의 발생(throwing)이라고 한다. 응용프로그람에서 오유를 조종하기 위하여 설치하는 개별적인 코드를 례외처리부(exception handler) 또는 포착블로크(catch block)라고 한다. 이것은 성원함수에 의해 발생한 례외를 받아들인다. 그 클라스의 객체를 사용하는 응용프로그람의 일부 코드는 try블로크안에 들어있다. try블로크안에서 발생되는 오유는 포착블로크에서 받아들일수 있다. 그 클라스와 교제하지 않는 코드는 try블로크안에 있을 필요가 없다. 그림 14-3에서 그 배치를 보여준다.

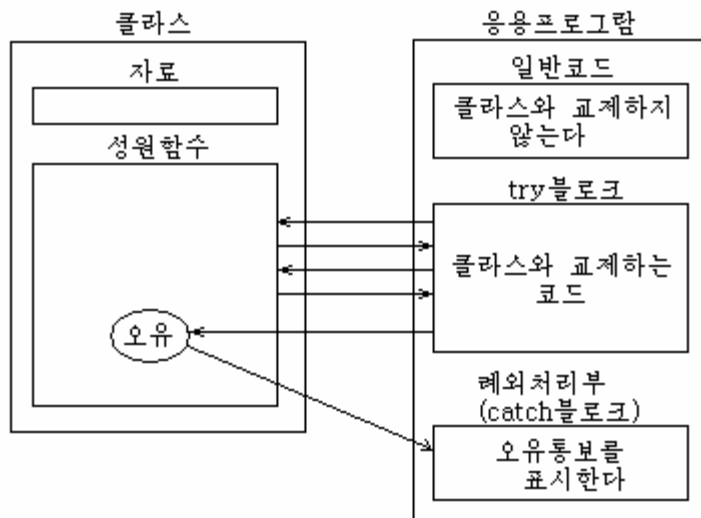



그림 14-3. 례외기구

례외기구는 C++의 새로운 예약어 throw, catch, try를 사용한다. 또한 례외클라스라고 부르는 새로운 종류의 실례를 창조해야 한다. 실례 14-7은 동작하는 프로그람이 아니지만 그 문법을 보여주는 골격프로그람이다.

(실례 14-7) 형판로 실현한 련결목록

```
#include <iostream>
using namespace std;
class AClass
{
private:
   class AnError
   {};
   void Func()
   {
      if(/* 오유조건 */)
           throw AnError();     // 례외발생
   }
```

```
};
int main()
{
  try// Try블로크
  {
    AClass obj;
    obj1.Func();
  }
  while(AClass::AnError) // 례외처리부
  {
    // 오유를 통보
  }
  return 0;
}
```

AClass라는 클라스로 시작한다. AClass클라스는 오유가 발생시킬수 있는 클라스를 보유한다. 례외클라스 AnError는 AClass의 공개부에 지정된다. AClass의 성원함수들에서는 오유를 검사한다. 오유를 발견하면 오유클라스용 구성자가 뒤에 오는 예약어 throw에 의해 례외를 발생시킨다. 즉

```
throw AnError(); // 례외발생
```

main()부분에서 try블로크에서 AClass와 교제하는 명령문들을 괄호에 넣어야 한다. 그중 임의의 명령문은 AClass성원함수안에서 오유를 검색하게 하고 례외를 발생시키며 즉시 try블로크뒤에 오는 catch블로크로 넘어가게 한다.

3. 간단한 례외실례

례외를 사용하는 프로그람의 실례를 들어보자. 이 실례는 실례 7-8을 변경한것이다. 여기서 탄창에는 옹근수자료를 보관할수 있다. 처음의 실례는 두개의 일반오유를 탐색하지 못한다. 응용프로그람이 탄창에 객체를 너무 많이 밀어넣으려고 하면 배렬의 용량을 초과하게 된다. 또한 빈 탄창에서 꺼내려고 하면 무효한 자료를 얻을수 있다. 실례 14-8에서는 례외에 의하여 두개의 오유를 조종한다.

(실례 14-8) 례외

```
#include <iostream>
using namespace std;
const int MAX = 100;
class Stack
{
private:
  int st[MAX]; int top;
public:
  class Range {};     // 례외클라스
  Stack() { top = -1; }
  void Push(int var)
  {
    if(top >= MAX-1) throw Range();    // 례외의 발생
```

```
        st[++top] = var;
    }
    int Pop()
    {
        if(top < 0) throw Range();
        return st[top--];    //  례외의 발생
    }
};
int main()
{
    Stack s1;
    try
    {
        s1.Push(11);
        s1.Push(22);
        s1.Push(33);
//  s1.Push(44);
        cout << "1: " << s1.Pop() << endl;
        cout << "2: " << s1.Pop() << endl;
        cout << "3: " << s1.Pop() << endl;
        cout << "4: " << s1.Pop() << endl;
    }
    catch(Stack::Range)
    {
        cout << "례외: 현재 탄창이 비거나 꽉찼습니다." << endl;
        return 0;
    }
    cout << "catch후에 여기로 넘어왔습니다." << endl;
    return 0;
}
```

너무 많은 항목을 밀어넣음으로써 례외가 발생할수 있게 탄창을 작게 만들었다.

례외를 론하는 이 프로그람의 특성이 4가지 있다. 클라스지정에는 례외클라스가 있다. 또한 례외를 발생시키는 명령문도 있다. main()부분에는 례외를 일으키는 코드블로크(try블로크)가 있고 례외를 조종하는 코드블로크(catch블로크)가 있다.

1) 례외클라스지정

프로그람은 우선 Stack클라스안에 례외클라스를 지정한다.

```
class Range
{
};
```

여기서 클라스본체는 비였으므로 클라스의 객체에는 자료도 없고 성원함수도 없다. 여기서 필요한것은 클라스이름 Range이다. 이 이름은 throw명령문과 catch블로크를 련결하는데 사용된다.(클라스본체가 항상 비는것은 아니다.)

Stack클라스에서 례외는 응용프로그람이 탄창이 비였는데 값을 꺼내려고 하거나 탄창이 다 찼는데 값을 밀어넣으려고 할 때 발생한다. Stack객체를 조작할 때 그러한

오유가 발생하였다는것을 응용프로그람에 알리기 위하여 Stack클라스의 성원함수들에서 if명령문에 의하여 조건을 검사하고 오유가 발생하면 례외를 발생시킨다. 실례 14-8에서 례외는 명령문 throw Range();에 의하여 두곳에서 발생한다.

명령문의 Range()부분은 Range클라스의 객체를 창조하는 구성자를 암시적으로 호출한다. 명령문의 throw부분은 프로그람의 조종을 례외처리부에로 옮긴다.

2) try블로크

이 례외를 일으키는 main()의 모든 명령문 즉 Stack객체를 조작하는 명령문들은 괄호안에 들어있고 try예약어뒤에 놓인다.

```
try
{
  // 례외를 일으키는 객체들에 대하여 조작하는 코드
}
```

이것은 응용프로그람의 단순한 일반코드부분이고 례외를 사용하지 않으면 쓸 필요가 없다. 프로그람의 모든 코드를 try블로크안에 넣을 필요는 없고 Stack클라스와 교제하는 코드만 넣어야 한다. 또한 프로그람에 try블로크가 많을수 있고 다른 곳에서 Stack객체들을 호출할수 있다.

3) 례외처리부(catch블로크)

례외를 조종하는 코드는 괄호안에 있고 catch예약어뒤에 놓인다. 또한 catch뒤의 괄호안에는 례외클라스이름이 있다. 례외클라스이름은 그것이 위치하는 클라스를 포함해야 한다. 여기서는 Stack:: Range.

```
catch(Stack::Range)
{
    // 례외를 조종하는 코드
}
```

이 구성을 례외처리부라고 부른다. 조종은 즉시 try블로크에 넘어간다. 실례 14-8에서 례외처리부는 단순히 오유통보를 출력하여 사용자에게 프로그람이 실패한 원인을 알려준다.

조종이 례외처리부의 아래로 내려오면 그로부터 처리를 계속할수 있다. 또한 례외처리부는 조종을 다른 곳에 넘기거나 프로그람을 완료할수 있다.

4) 사건의 렬

례외가 발생할 때 사건들의 렬을 요약한다.

① 코드는 try블로크밖에서 정상적으로 실행되고있다.

② 조종이 try블로크에 들어온다.

③ try블로크안의 명령문은 성원함수에서 오유를 일으킨다.

④ 성원함수는 례외를 발생시킨다.

⑤ 조종은 try블로크 뒤의 례외처리부(catch블로크)로 넘어간다.

try블로크안에 있는 임의의 명령문이 례외를 발생시킬수 있으나 try-throw-catch

배렬이 그것을 모두 자동적으로 조종하므로 매개의 돌림값을 검사할 필요는 없다. 이 실례에서는 례외를 일으키는 명령문이 두개 있다. 첫째로

```
s1.Push(44);
```

은 그 앞에 놓인 설명문을 지우면 오유를 일으킨다.

둘째로

```
cout << "4: " << s1.Pop() << endl;
```

은 첫 명령문이 실행문으로 되여있으면 례외를 일으킨다. 두 경우에 같은 오유통보 《례외: 현재 탄창이 비거나 꽉 찼습니다.》가 표시된다.

4. 다중례외

필요한만큼 례외를 많이 발생시키도록 클라스를 설계할수 있다. 이것을 보여주기 위하여 실례 14-8의 프로그람에서 탄창에 자료를 밀어넣을 때의 례외와 빈 탄창에서 자료를 꺼내려고 할 때의 례외를 따로따로 발생시키도록 변경한다.

(실례 14-9) 두개의 례외처리부

```
#include <iostream>
using namespace std;
const int MAX = 3;
class Stack
{
private:
   int st[MAX];
   int top;
public:
   class Full {};
   class Empty {};
   Stack() { top = -1; }
   void Push(int var)
   {
      if(top >= MAX-1)
          throw Full();    // 례외의 발생
      st[++top] = var;
   }
   int Pop()
   {
      if(top < 0)
          throw Empty();
      return st[top--];   //  례외의 발생
   }
};
int main()
{
   Stack s1;
   try
   {
```

```
        s1.Push(11);
        s1.Push(22);
        s1.Push(33);
//      s1.Push(44);
        cout << "1: " << s1.Pop() << endl;
        cout << "2: " << s1.Pop() << endl;
        cout << "3: " << s1.Pop() << endl;
        cout << "4: " << s1.Pop() << endl;
    }
    catch(Stack::Full)
    {
        cout << "례외: 현재 탄창이 다 찼습니다." << endl;
        return 0;
    }
    catch(Stack::Empty)
    {
        cout << "례외: 현재 탄창이 비였습니다." << endl;
        return 0;
    }
    return 0;
}
```

실례 14-9에서는 두개의 례외클라스를 지정한다.

```
class Full {};
class Empty {};
```

명령문 throw Full();은 응용프로그람이 탄창이 다 찼는데 Push()를 호출할 때 실행되고 throw Empty();는 탄창이 비였는데 Pop()를 호출하면 실행된다.

개별적인 catch블로크가 매개의 례외에 사용된다. 즉

```
try
{
  // Stack객체에 조작하는 코드
}
catch(Stack::Full)
{
  // Full례외조종코드
}
catch(Stack::Empty)
{
  // Empty례외조종코드
}
```

특정한 try블로크를 사용하는 모든 catch블로크는 try블로크뒤에 놓여야 한다. 이 경우에 매개 catch블로크는 단순히 통보문 《례외: 현재 탄창이 다 찼습니다.》 또는 《례외: 현재 탄창이 비였습니다.》를 출력한다. 주어진 례외에 대하여 오직 하나의 catch블로크가 실행된다. catch블로크묶음(catch ladder)은 실행되는 적당한 코드절을 가지는 switch명령문과 거의 비슷하다. 례외를 조종했을 때 조종은 모든 catch블로크

뒤에 오는 명령문에 넘어간다.(switch명령문과 달리 break를 사용하여 catch블로크를 완료할 필요가 없다. 이 방법에서 catch블로크는 함수와 더 류사하게 동작한다.)

5. Distance클라스에서 례외

례외의 다른 실례를 고찰하자. 이것은 앞에서 언급한 Distance클라스에 례외를 적용한것이다. Distance객체는 옹근수값 메터와 류동소수점수값 센치메터를 가진다. 센치메터값은 항상 100.0보다 작아야 한다. 앞의 실례에서 Distance클라스와 관련한 문제는 사용자가 100.0이상의 값을 가지는 센치메터로 객체를 초기화한 경우에 그것을 탐색할수 없는것이다. 이것은 센치메터가 100.0보다 작다고 가정한 산수루틴(operator+ 등)에 의해 산수연산하려고 할 때 난관으로 된다. 그러한 불가능한 값들을 표시할수도 있으므로 사용자가 7m 115cm와 같은 크기를 입력하지 않도록 하여야 한다.

이 오유를 조종하는데 례외를 사용하도록 Distance클라스를 수정한다.

(실례 14-10) Distance클라스에서 례외

```
#include <iostream>
using namespace std;
class Distance
{
private:
   int meters;
   float centies;
public:
   class CentiesEx {};
   Distance() : meters(0), centies(0.0) {}
   Distance(int me, float ce)
   {
      if(centies >= 100.0)
          throw CentiesEx();
      meters = me;
      centies = ce;
   }
   void GetDist()
   {
      cout << "\n메터를 입력하시오:";
      cin >> meters;
      cout << "센치메터를 입력하시오:";
      cin >> centies;
      if(centies >= 100.0)
          throw CentiesEx();
   }
   void ShowDist()
   {
      cout << meters << "m " << centies << "cm";
```

```
    }
};
int main()
{
    try
    {
        Distance dist1(17, 3.5);
        Distance dist2;
        dist2.GetDist();
        cout << "\ndist1 = ";
        dist1.ShowDist();
        cout << "\ndist2 = ";
        dist2.ShowDist();
    }
    catch(Distance::CentiesEx)
    {
        cout << "\n초기화오유: 센치메터값이 너무 큽니다.";
    }
    cout << endl;
    return 0;
}
```

Distance클라스에 CentiesEx라는 례외클라스를 설치한다. 그다음 사용자가 100.0과 같거나 큰 값으로 센치메터자료를 초기화하면 례외를 발생시킨다. 례외는 두곳에서 발생한다. 2인수구성자에서는 프로그람작성자가 초기값을 제공하면서 오유를 발생시킬수 있고 GetDist()함수에서는 사용자가 "센치메터를 입력하시오."재촉문에서 옳지 않은 값을 입력할 때 발생한다. 또한 부수값과 다른 입력오유도 검사할수 있다.

main()에서 Distance객체와의 모든 교제는 try블로크에 들어있고 catch블로크는 오유통보를 표시한다.

물론 더 복잡한 프로그람에서 사용자오유를 각이하게 조종하려고 할수 있다. 실례로 try블로크의 선두로 가서 사용자에게 다른 거리값을 입력할 기회를 주는것이 더 좋다.

6. 인수를 가지는 례외

무엇이 례외를 일으켰는가에 대한 정보를 응용프로그람이 요구한다면 어떻게 하겠는가?

실례 14-4에서는 부정확한 센치메터값이 실제로 무엇인가를 알수 있도록 프로그람작성자를 방조할수 있다. 또한 실례 14-10처럼 같은 례외가 다른 성원함수에 의해 발생하면 어느 함수가 원인으로 되는가를 알아야 한다.

그러면 례외가 발생한 성원함수로부터 그것을 포착하는 응용프로그람에로 그러한 정보를 넘기는 방법이 있는가?

례외발생이 례외처리부에로 조종을 넘길뿐아니라 례외클라스의 구성자를 호출하여

그 객체를 창조함으로써 여기에 대응할수 있다.

실례 14-10에서 명령문 throw CentiesEx();에 의해 례외를 발생시킬 때 CentiesEx형의 객체를 창조한다.

례외클라스에 자료성원을 추가하면 객체를 창조할 때 그것들을 초기화할수 있다. 그다음 례외처리부가 례외를 포착하여 객체로부터 자료를 얻을수 있다. 이것이 가능하도록 실례 14-10을 수정하였다.

(실례 14-11) 인수를 가지는 례외

```
#include <iostream>
#include <string>
using namespace std;
class Distance
{
private:
  int meters;
  float centies;
public:
  class CentiesEx
  {
  public:
     string origin;
     float iValue;
     CentiesEx(string or, float ce)
     {
         origin = or;
         iValue = ce;
     }
  };
  Distance() : meters(0), centies(0.0) {}
  Distance(int me, float ce)
  {
     if(centies >= 100.0)
         throw CentiesEx("2인수 구성자", ce);
     meters = me;
     centies = ce;
  }
  void GetDist()
  {
     cout << "\n메터를 입력하시오:";
     cin >> meters;
     cout << "센치메터를 입력하시오:";
     cin >> centies;
     if(centies >= 100.0)
         throw CentiesEx("GetDist()함수", centies);
  }
  void ShowDist()
```

```
    {
        cout << meters << "m " << centies << "cm";
    }
};
int main()
{
    try
    {
        Distance dist1(17, 3.5);
        Distance dist2;
        dist2.GetDist();
        cout << "\ndist1 = ";
        dist1.ShowDist();
        cout << "\ndist2 = ";
        dist2.ShowDist();
    }
    catch(Distance::CentiesEx ix)
    {
        cout << endl << ix.origin << "에서 초기화오유: "
             << ix.iValue << "의 값이 너무 큽니다.\n";
    }
    cout << endl;
    return 0;
}
```

례외를 발생시킬 때 자료를 넘기는 조작에는 세개 부분 즉 례외클라스용의 자료성원들과 구성자를 지정하는 부분, 례외를 발생시킬 때 례외객체를 초기화하는 부분, 례외를 받아들일 때 객체의 자료를 호출하는 부분이 있다. 이것들을 차례로 고찰해보자.

1) 례외클라스안의 자료지정

례외클라스의 자료는 례외처리부에 의하여 그것을 직접 호출할수 있도록 공개로 하는것이 관례이다. 여기에 실례 14-11의 새로운 CentiesEx례외클라스가 있다.

```
class CentiesEx
{
public:
    string origin;
    float iValue;
    CentiesEx(string or, float ce)
    {
        origin = or;
        iValue = ce;
    }
};
```

례외객체에는 두개의 공개변수 즉 호출되고있는 성원함수의 이름을 보관하는 string객체와 부정확한 센치메터값을 보관하기 위한 float형변수가 있다.

2) 례외객체의 초기화

그러면 례외를 발생시킬 때 자료를 어떻게 초기화하는가?

Distance클라스용의 2인수구성자에서는 다음과 같이 례외객체를 창조한다.

```
throw CentiesEx("2인수 구성자", ce);
```

Distance의 GetDist()성원함수에서는 다음과 같이 례외객체를 창조한다.

```
throw CentiesEx("GetDist()함수", centies);
```

례외가 발생할 때 례외처리부는 문자렬과 센치메터값을 표시한다. 문자렬은 어느 함수가 례외를 발생시키는가를 알려주고 성원함수가 발견한 부정확한 센치메터값을 알린다. 이 수값자료는 프로그람작성자에게 오유가 발생한 원인을 알려준다.

7. 례외객체로부터 자료의 얻기

그러면 례외를 받아들일 때 자료를 어떻게 얻겠는가?

가장 간단한 방법은 이 실례처럼 자료를 례외클라스의 공개부분으로 만드는것이다. 그다음 포착블로크에서 ix를 받아들이는 례외객체의 이름으로 선언한다. 이름과 함께 점연산자를 사용하는 일반적인 방법으로 그 자료를 참고한다.

```
catch(Distance::CentiesEx ix)
{
   cout << endl << ix.origin << "에서 초기화오유: "
          << ix.iValue << "의 값이 너무 큽니다.\n";
}
```

그다음 ix.orign과 ix.ivalue값을 표시할수 있다. 여기에 사용자가 매우 큰 센치메터값을 입력할 때 실례 14-11과의 대화가 있다.

```
메터를 입력하시오: 17
센치메터를 입력하시오: 120.3
GetDist()함수에서 초기화오유: 120.3의 값이 너무 큽니다.
```

마찬가지로 작성자가 main()에서 dist1의 정의를 Distance dist1(17,122.2f);로 변경한다면 례외는 다음의 오유통보를 내보낸다.

```
2인수 구성자에서 초기화오유: 120.3의 값이 너무 큽니다.
```

물론 례외인수를 필요한만큼 만들수 있으나 일반적으로 례외를 일으킨 오유를 진단하는데 도움이 되는 정보를 보유한다.

8. bad_alloc클라스

표준 C++에는 여러개의 기본례외클라스가 있다. 제일 많이 사용하는것이 bad_alloc이다. 이것은 new로 기억할당하려고 할 때 오유가 생기면 례외를 발생시킨다.(이 례외는 C++의 이전판에서 xalloc라고 불렀다. 이 수법은 여전히 Visual C++에서 쓰이고있다.) 적당한 try와 catch블로크를 설정하면 bad_alloc를 사용할수 있다.

(실례 14-12)　bad_alloc례외

```
#include <iostream>
using namespace std;
```

```
int main()
{
   const unsigned long SIZE = 10000;
   char* ptr;
   try
   {
      ptr = new char[SIZE];
   }
   catch(bad_alloc)
   {
      cout << "\nbad_alloc례외: 기억기를 할당할수 없습니다.\n";
      return 1;
   }
   delete[] ptr;
   cout << "\n기억기를 성과적으로 할당하였습니다.\n";
   return 0;
}
```

9. 례외에서 주의사항

가장 간단하고 일반적인 례외사용방법을 보았다. 례외사용법에 대하여 몇가지 알아야 할것이 있다.

1) 함수겹쌓임

례외를 일으키는 명령문은 try블로크에 직접 배치되여있을뿐아니라 try블로크의 명령문에 의해 호출되는 함수안에 있거나 try블로크안의 명령문에 의하여 호출되는 함수에서 호출되는 함수안에 있을수 있다. 따라서 프로그람의 웃부분에 try블로크들을 설치해야 한다. 아래준위의 함수들은 try블로크안의 함수에 의해 직접 혹은 간접적으로 호출되게 한다.

2) 례외와 클라스서고

례외에 의하여 해결되는 중요한 문제는 클라스서고의 오유문제이다. 서고루틴은 오유를 발견할수 없다. 즉 오유에 대하여 모른다. 대체로 서고루틴은 서로 다른 사람에 의해, 프로그람이 호출하는 시간도 각이한 조건에서 사용된다. 서고루틴이 해야 할 일은 프로그람이 그것을 호출할 때 항상 그에 대한 오유를 넘기는것이다. 말하자면 호출측프로그람은 이렇게 오유를 조절할수 있다. 례외기구는 례외가 catch블로크와 만날때까지 겹쌓인 함수들을 통하여 우로 전달되게 한다. throw명령문은 서고루틴안에 있을수 있다. catch블로크는 오유를 론하는 방법을 아는 프로그람에 있을수 있다.

만일 클라스서고를 쓰고있다면 그것을 사용하는 프로그람에 문제를 일으킬수 있는 것들에 대하여 례외를 발생시키게 해야 한다. 클라스서고를 사용하는 프로그람을 쓴다면 그것이 발생시키는 례외를 위한 try와 catch블로크를 제공하여야 한다.

3) 모든 경우에 례외를 사용해서는 안된다.

례외는 모든 종류의 오유에 사용하지 말아야 한다.

례외는 프로그람을 실행(례외발생)할 때 일정한 시간을 소비한다.

실례로 례외는 프로그람에 의해 쉽게 탐색할수 있는 사용자의 오유에는 사용하지 말아야 한다.

4) 자동적으로 호출되는 해체자

례외기구는 아주 복잡하다. 례외가 발생할 때 해체자는 try블로크안의 그전까지의 코드에 의하여 생성되는 객체에 대하여 자동적으로 호출된다. 이것은 어느 명령문이 례외를 일으켰는지 응용프로그람이 모르기때문에 요구된다. 또한 오유로부터 복귀하려면 try블로크의 웃끝으로부터 다시 시작해야 한다. 례외기구는 객체들사이의 관계가 복잡해지면 최소한 try블로크안의 코드를 재설정한다.

5) 례외조종(handling exception)

례외를 받아들인 다음 대체로 응용프로그람을 완료한다. 례외기구는 사용자에게 오유의 원천을 지적할 기회를 주고 완료하기 전에 필요한 해체작업을 수행할 기회를 준다. 또한 try블로크안에서 창조된 객체용의 해체자를 실행함으로써 해체를 더 쉽게 한다. 이것은 체계자원(즉 객체가 사용하고있는 기억기 등)을 해방하게 한다.

다른 경우에는 프로그람을 완료하지 않고 무엇이 오유를 일으켰으며 오유를 어떻게 수정하겠는가를 고찰한다. 또한 사용자가 다른 자료를 입력하게 할수 있다. 그러한 경우에 try와 catch블로크는 대체로 순환에 들어가므로 조종은 try블로크의 선두에로 돌아갈수 있다. (한편 례외기구는 그 초기상태로 복귀하려고 한다.)

발생한 례외와 일치하는 례외처리부가 없으면 프로그람은 조작체계에 의하여 강제로 완료한다.

요 약

형판은 함수족 또는 클라스족을 형성한다. 즉 각이한 자료형을 조종하게 한다. 서로 다른 자료형에 대하여 같은 조작을 수행하는 등가한 함수를 여러개 쓰려고 한다면 그 대신 형판함수를 사용한다. 마찬가지로 자료의 형만 다른 여러개의 각이한 클라스 명세를 쓰려고 한다면 클라스형판을 사용할수 있다.

례외는 계층적인 객체지향수법으로 C++오유를 조종하는 기구이다. 례외는 자체로 클라스의 객체들에 대하여 조작하는 try블로크안의 오유가 있는 명령문에 의하여 발생한다. 클라스성원함수는 오유를 발견하고 례외를 발생시키며 try블로크뒤의 례외처리부코드에서 그 클라스를 사용하는 프로그람에 의해 받아들인다.

문 제

1. 형판은

① 변수들

② 함수들

③ 클라스들

④ 프로그람들의 계렬을 만드는데 편리한 방법을 제공한다.

어느것이 옳은가?

2. 형판인수는 어떤 예약어의 뒤에 놓이는가?

3. 형판은 각이한 판의 함수들을 사용자의 입력에 따라 자동적으로 창조한다. 옳은가?

4. 인수의 두배를 돌려주는 함수형판을 쓰시오.

5. 형판클라스는

① 각이한 용기에 보관하기 위해 설계된다.

② 각이한 자료형과 작업한다.

③ 모두 등가한 객체들을 생성한다.

④ 각이한 개수의 성원함수들을 가지는 클라스들을 생성한다.

어느것이 옳은가?

6. 형판인수는 한개이상일수 있다. 옳은가?

7. 형판로부터 실제함수를 창조하는것을 무엇이라고 하는가?

8. 형판함수의 실제코드는

① 함수선언이 원천코드에 나타날 때

② 함수정의가 원천코드에 나타날 때

③ 함수호출이 원천코드에 나타날 때

④ 함수가 실행시에 실행될 때 생성된다.

어느것이 옳은가?

9. 형판의 기본개념은 고정자료형을 어떤 자료형에 대하여 선언하는 이름으로 교체하는것이다.

10. 형판은 일반적으로 어떤 클라스들에 사용한다.

11. 례외는 전형적으로

① 응용프로그람의 코드를 쓰는 프로그람작성자

② 클라스성원함수들을 쓰는 클라스작성자

③ 실행시오유

④ 프로그람을 완료하는 조작체계제작자에 의해 발생된다.

어느것이 옳은가?

12. 례외와 함께 쓰이는 C++예약어들은 무엇인가?

13. 빈 본체를 가지는 클라스 BoundsError를 사용하여 례외를 발생시키는 명령문을 쓰시오.

14. 례외를 일으킬수 있는 명령문들은 catch블로크의 부분이여야 한다. 옳은가?

15. 례외는

① catch블로크로부터 try블로크로 투입된다.

② throw명령문으로부터 try블로크로 투입된다.

③ 오유점으로부터 catch블로크로 투입된다.

④ throw명령문으로부터 catch블로크로 투입된다. 어느것이 옳은가?

16. 오유번호와 오유이름을 보관하는 례외클라스를 선언하시오. 구성자를 포함하시오.

17. 례외를 발생하는 명령문은 try블로크에 배치할 필요가 없다. 옳은가?

18. 다음것은 례외가 전형적으로 발생되는 오유이다.

① 너무 많은 량의 자료는 배렬을 넘쳐나게 할 위험이 있다.

② 사용자는 Ctrl+C건을 눌러서 프로그람을 완료한다.

③ 치명적인 오유는 체계의 전원을 차단하게 한다.

④ new는 요구하는 기억기를 얻을수 없다.

어느것이 옳은가?

19. 례외가 발생할 때 보내오는 추가정보는

① throw예약어

② 오유를 일으킨 함수

③ catch블로크

④ 례외클라스객체안에 있을수 있다.

어느것이 옳은가?

20. 프로그람은 례외가 발생한 후에도 계속 동작할수 있는가?

련습문제

1. 배렬의 모든 원소들의 평균값을 돌려주는 형판함수를 쓰시오. 함수에로의 인수들은 배렬이름과 배렬크기(int형)일수 있다. main()에서 int, long, double, char*형의 배렬을 가진 함수를 시험하시오.

2. 대기렬은 자료보관용기이다. 탄창과 비슷하지만 LIFO대신 FIFO(First-In First-Out)이며 은행에서 계산원의 창구앞에서 기다리는 손님들의 줄과 비슷하다. 1,2,3을 입력하면 그 순서로 1,2,3이 얻어진다. 탄창은 배렬에 대한 한개 첨수를 요구한다. 다른 한편 대기렬은 배렬에 대한 두개 첨수들의 리력을 유지해야 한다. 즉 하나는 새로운 항목이 추가되는 꼬리부를 가리키는 첨수, 또 하나는 이전 항목을 삭제하는 머리부를 가리키는 첨수. 꼬리부는 마디나 항목이 추가, 삭제될 때 배렬을 지나게 된다. 꼬리부나 머리부가 배렬의 끝에 이르면 그것을 배렬의 선두로 다시 설정한다. 대기렬클라스

의 클라스형판을 쓰시오. 대기렬을 사용하는 사용자가 오유를 일으키지 않는다고 가정하자. 즉 대기렬의 용량을 초과하지 않으며 또는 대기렬이 빌 때 항목을 삭제하려고 하지도 않는다. 각이한 자료형의 대기렬을 여러개 정의하고 거기에 자료를 삽입하고 그로부터 자료를 삭제하시오.

3. 련습 2의 대기렬형판에 례외를 추가하시오. 두개의 례외를 고찰하자. 하나는 대기렬의 용량이 넘어난 경우이고 다른 하나는 프로그람이 빈 대기렬로부터 항목을 지우려고 하는 경우이다. 이것을 조작하는 방법은 대기렬에 새로운 자료성원을 추가하는 것이다. 즉 대기렬에서 현재 항목들을 계수하는것이다. 한개 항목을 삽입할 때 계수값(count)은 증가하며 한개 항목을 삭제할 때 그 값은 감소한다. 계수값이 대기렬의 용량을 초과하거나 0보다 작아지면 례외가 발생한다. main()프로그람에서 사용자가 대기렬에 값들을 넣고 그것을 얻을수 있게 하시오. 례외에 의하여 프로그람은 대기렬의 내용을 잃지 않고 오유로부터 복귀해야 한다.

4. 넘어온 두개의 인수값을 호상 교환하는 Swap()함수를 정의하시오. 함수를 형판로 만들어 모든 수값자료형(char, int, float,…)에 사용할수 있게 하시오. main()프로그람에서 여러 형에 대하여 함수를 시험하시오.

5. 배렬의 최대원소값을 돌려주는 AMax()함수를 창조하시오. 함수의 인수들은 배렬의 주소와 크기이다. 이 함수를 형판로 만들어 임의의 수값자료형과 작업할수 있게 하시오. 여러가지 형의 배렬에 이 함수를 적용하는 main()프로그람을 작성하시오.

6. 실례 8-12의 SafeArray를 사용하시오. 이 클라스를 형판로 만들어 안전한 배렬이 임의의 형의 임의의 배렬자료를 보관할수 있게 하시오. main()에서 적어도 두개의 다른 자료형의 안전한 배렬을 창조하고 거기에 자료를 보관하시오.

7. 8장 련습 7의 Fraction클라스와 4기능분수수산기를 사용하시오. Fraction클라스를 형판로 만들어 나누이는 수와 나누는 수에 대하여 각이한 자료형을 사용하여 실례화할수 있게 하시오. 이것들은 char, short, int, long과 같은 옹근수형이여야 한다. main()에서 클라스 Fraction<char>를 실례화하고 4기능분수수산기에서 그것을 사용하시오. 클라스 Fraction<char>는 Fraction<int>보다 적은 기억기를 취하지만 큰 분수를 기억할수 있다.

8. 실례 8-12에 례외클라스를 추가하여 경계를 벗어나는 경우에 례외를 발생하게 하시오. catch블로크는 사용자에게 오유를 통보해야 한다.

9. 련습 8에서 례외클라스를 수정하여 catch블로크안의 오유통보가 례외를 일으키는 첨수값을 통보하게 하시오.

10. 실례 12-1을 참고하시오. 사용자의 입력오유를 례외로 가정하시오. 프로그람의 Distance클라스에 례외클라스를 추가하시오. (실례 14-10과 14-11을 참고하시오.) 실례 12-1이 오유통보문을 표시하는 모든 곳에서 례외를 발생시키시오. 례외구성자에로의 인수를 사용하여 오유가 발생한 곳과 오유원인(수가 아닌 인수, 범위밖의 인수

등)을 통보하시오. 또한 IsInt()함수에서 오유가 발생할 때(아무것도 입력하지 않을 때, 너무 많은 수자를 입력할 때, 수자가 아닌 문자를 입력할 때, 옹근수범위를 벗어날 때) 례외를 발생시키시오.

11. 실례 8-6을 고찰하시오. 례외클라스를 추가하고 초기화문자렬이 너무 길면 1인수구성자에서 례외를 발생시키시오. 또한 두개의 문자렬을 련결할 때 결과가 너무 길면 재정의된 +연산자에서 례외를 발생시키고 어떤 오유가 발생하였는가를 통보하시오.

12. 흔히 례외를 사용하는 가장 간단한 방법은 례외클라스가 성원인 새로운 클라스를 창조하는것이다. 파일오유를 조종하는데 례외를 사용하는 클라스를 시험하시오. 례외클라스를 포함하며 파일을 읽고 쓰는 성원함수들을 포함하는 Dofile클라스를 만드시오. 이 클라스의 구성자는 파일이름을 인수로 가지며 그 이름으로 파일을 연다. 또한 하나의 성원함수는 파일지적자를 파일의 선두로 재설정한다. 실례 12-14를 참고하시오. main()프로그람에서 이 기능을 시험해보시오.

제 15 장. 표준형판서고

대부분의 콤퓨터프로그람은 자료를 처리하기 위하여 존재한다. 자료는 련관된 현실세계정보 즉 개인자료레코드, 명세목록, 본문문서, 과학실험결과 등을 표시할수 있다. 자료가 표시하는것이 무엇이든 자료는 기억기에 보관되고 류사한 방법으로 조작된다. 《자료구조와 알고리듬》이라는 학과목에서 자료구조는 기억기에 자료를 보관하는 방법을 취급하고 알고리듬은 그것을 조작하는 방법을 서술한다.

C++클라스들은 자료구조의 서고를 창조하기 위한 우수한 기구를 제공한다. 이미전에 번역프로그람제공단위들과 제3부류 개발자들은 자료의 보관과 체계를 조종하기 위한 용기클라스의 서고를 제공해왔다. 그러나 현재 표준 C++는 용기서고를 포함하고있다. 그것을 표준형판서고(Standard Template Library, 요약하여 STL)라고 한다. STL은 표준 C++클라스서고의 한부분으로서 자료를 보관하고 처리하는 표준수법으로 사용할수 있다.

이 장에서는 STL과 그 사용방법을 서술한다. STL은 크고 복잡하므로 그 매개에 대하여 구체적으로 설명하지 않는다. 여기서는 STL을 소개하고 일반적인 알고리듬과 용기의 실례를 준다.

제 1 절. STL의 개요

STL은 몇가지 종류의 실체를 포함한다. 그중 세가지 중요한것이 용기, 알고리듬, 반복자이다.

용기(container)는 보관하는 자료를 기억기에서 조직화하는 방법이다. 앞에서 두 종류의 용기 즉 탄창과 련결목록을 설명하였다. 다른 용기로서 배렬은 C++에 이미 짜넣어져있다. 그밖에도 많은 종류의 용기가 있다. STL은 가장 쓸모있는것들을 포함한다. STL용기는 형판클라스로 실현되므로 각종 자료를 유지하도록 간단히 전용화할수 있다.

STL에서 알고리듬(algorithm)은 여러가지 방법으로 용기의 자료를 처리하기 위하여 용기에 적용되는 수속이다. 실례로 자료의 정렬과 복사, 검색, 결합알고리듬이 있다. 알고리듬은 형판함수로 표시된다. 형판함수는 용기클라스의 성원함수가 아니라 독립적인 일반함수이다. STL의 중요한 특성의 하나는 알고리듬의 일반성이다. 알고리듬은 STL용기뿐아니라 일반적인 C++배렬과 자기가 창조한 용기들에 대하여 사용할수 있다. 또한 용기는 더 특정한 일감을 수행하기 위한 성원함수들을 포함한다.

반복자(iterator)는 지적자개념의 일반화이다. 즉 반복자는 용기안의 원소들을 지적한다. 지적자처럼 반복자를 증가시킬수 있으므로 용기안의 매개 원소들을 차례로 가리킬수 있다. 반복자는 알고리듬을 용기와 련결할수 있으므로 STL의 유효한 부분이다.

음성요소들을 모두 련결하는 케블, 또는 콤퓨터를 그 주변장치와 련결하는 케블의 쏘프트웨어판으로 반복자를 고찰할수 있다.

그림 15-1은 STL의 기본구성요소들을 보여준다. 이 절에서는 용기, 알고리듬, 반복자를 구체적으로 설명한다. 다음 절들에서는 프로그람실례를 통하여 이 개념들을 연구한다.

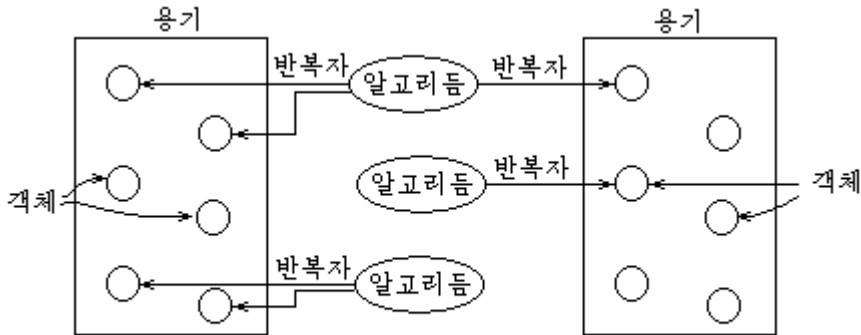



그림 15-1. 용기와 알고리듬, 반복자

1. 용기

용기는 자료를 보관하는 방법이다. 자료는 int, float형과 같은 기본자료형 또는 클라스의 객체들로 이루어진다. STL은 7가지 기본종류의 용기와 그로부터 파생되는 3가지 용기를 가능하게 한다. 또한 기본종류의 용기에 기초하여 자체의 용기도 창조할수 있다.

그러면 왜 그렇게 많은 종류의 용기가 필요한가?

그리고 모든 자료보관에 왜 C++배렬을 사용하지 않는가?

그것은 효과성에 있다. 배렬은 많은 경우에 불편하고 느리다.

STL의 용기는 두가지 부류 즉 순차용기와 련상용기로 나눌수 있다. 순차용기에는 vector, list, deque가 있다. 련상용기에는 set, multiset, map, multimap가 있다. 또한 3가지 특수용기 즉 stack, queue, priority_queue가 순차용기로부터 파생된다.

1) 순차용기

순차용기(sequence containers)는 일련의 원소들을 거리의 주택들처럼 차례로 볼수 있게 보관한다. 매개 원소는 행에서의 그 위치에 의하여 다른 원소와 련결된다. 마지막 원소를 제외한 매개 원소는 어떤 원소의 앞에 놓이고 다른 원소의 뒤에 놓인다. 보통 C++배렬은 순차용기의 실례이다.

C++배렬과 관련한 하나의 문제는 번역시 즉 원천코드에서 그 크기를 지적해야 하는것이다. 그런데 어떤 경우에는 프로그람을 쓸 때 배렬에 자료를 얼마나 보관할지 모르므로 자료의 최대량을 보관할수 있게 충분히 큰 배렬을 지정해야 한다. 프로그람을

실행할 때 배렬을 다 채우지 않으므로 기억기를 랑비하거나 공간밖에서 실행하여 오유통보를 낼수 있다. STL은 이러한 결함을 피하기 위하여 vector용기를 제공한다.

배렬과 관련한 또 다른 문제가 있다. 종업원의 레코드를 보관할 때 이름에 따라서 자모순으로 배렬하였다고 하자. 현재 이름이 L로 시작하는 새 용기를 삽입하려고 한다면 빈자리를 만들기 위하여 M으로부터 Z까지의 모든 종업원들을 모두 옮겨야 한다. 이것은 시간을 많이 소비한다. STL은 련결목록의 개념에 기초하는 list용기를 제공하고 list가 이 문제를 해결한다. 여러개의 지적자를 재배치하여 련결목록에 새로운 항목을 쉽게 삽입하는 실례 10-23을 참고하시오.

세번째 순차용기는 deque이다. deque는 탄창과 대기렬의 결합으로 생각할수 있다. 탄창은 LIFO원리에 따라 작업한다. 입출력은 모두 탄창의 꼭대기에서 발생한다. 다른 한편 대기렬은 FIFO배치를 사용한다. 따라서 자료는 은행에서 손님들의 렬처럼 앞으로 들어오고 뒤로 나간다. deque는 이 방법을 결합하여 다른 끝에 자료를 삽입하거나 그로부터 자료를 삭제할수 있다. deque라는 단어는 Double-Ended QUEue로부터 파생된것이다. 이것은 탄창과 대기렬의 기초로 사용할수 있는 만능기구이다.

표 15-1은 STL순차용기의 특성을 요약한다. 그것은 보통의 C++배렬을 포함한다.

표 15-1. 기본순차용기

구분	특성	우결함
보통의 C++배렬	고정크기	고속직접호출(첨수에 의해)할수 있다. 중간에서 삽입 혹은 삭제가 느리다. 실행시에 크기를 변경할수 없다.
vector	재배치, 확장가능한 배렬	첨수에 의한 고속집접호출할수 있다. 중간에서 삽입과 삭제가 느리다. 끝에서 고속삽입 혹은 고속삭제할수 있다.
list	2중련결목록	임의의 위치에서 고속삽입 혹은 삭제할수 있다. 량끝에서 고속호출할수 있다. 직접호출이 느리다.
deque	vector와 같은데 량끝에서 호출할수 있다.	첨수를 사용한 고속직접호출할수 있다. 중간에서 삽입과 삭제가 느리다. 선두나 끝에서 고속삽입, 삭제, 고속넣기와 꺼내기를 할수 있다.

STL용기의 실례화는 쉽다. 우선 적당한 머리부파일을 포함한다. 그다음 파라메터로서 보관하려는 객체의 종류가 있는 형판형식을 사용한다.

실례로

```
vector<int> aVect;                // int의 벡토르창조
```

또는

list<AirTime> departureList; // AirTime의 목록창조

STL용기의 크기를 지적할 필요는 없다. 용기자체가 모든 기억할당을 조종한다.

2) 련상용기

련상용기(associative containers)는 순차적이 아니다. 또 하나의 련상용기는 자료를 호출하기 위한 건(key)을 사용한다. 수자나 문자렬로 이루어지는 건들은 보관된 원소들을 특정한 순서로 배렬하고 용기에 의해 자동적으로 사용된다. 보통 조선말사전과 같다. 거기서 자모순으로 배렬된 단어들을 탐색함으로써 자료를 호출한다. 또한 건값으로 시작하는데 용기는 건을 기억기에서 그 원소의 배치로 전환한다. 건을 알면 련상값을 고속으로 호출할수 있다.

STL에는 두 종류의 련상용기 즉 set와 map가 있다. 이것들은 모두 tree라고 부르는 구조체에 자료를 보관한다. tree는 고속검색, 삽입, 삭제를 제공한다. set와 map는 광범한 응용프로그람에 적합한 아주 만능적인 일반자료구조이다. 그러나 그것들을 정렬하고 직접호출을 요구하는 다른 조작을 수행하는데서는 비효과적이다.

set는 map보다 단순하고 일반적으로 사용된다. set는 건을 포함하는 여러개의 항목을 보관한다. 건은 항목들을 정리하는데 쓰이는 속성이다. 실례로 set는 Person클라스의 객체들의 그 이름속성을 건에 기초하여 자모순으로 정렬하여 보관한다. 이 경우에 지정된 이름으로 객체를 검색함으로써 Person객체를 고속으로 배치한다. set가 int와 같은 기본형의 값들을 보관한다면 그것은 보관된 전체 항목이다. 일부 독자들은 건과 함께 목록에 보관된 전체 객체를 언급하지만 그것들을 정렬하는데 사용되는 속성이 전체항목이 아니라는것을 강조하기 위하여 건객체를 호출한다.

map는 객체의 쌍 즉 건객체와 값객체를 보관한다. map는 첨수에 의하여 원소를 호출할 대신에 임의의 형의 지시자(또는 색인)를 사용하여 호출한다는것을 제외하면 배렬과 같은 용기로서 사용할수 있다. 즉 건객체는 첨수에 해당되고 값객체는 그 첨수에 해당한 값이다.

map와 set용기에는 주어진 값에 대하여 오직 하나의 건이 보관된다. 이것은 종업원목록이 유일한 종업원번호에 따라 배렬되게 한다. 다른 한편 multimap와 multiset용기는 여러개의 건을 허용한다. 조선말사전에는 단어의 실체가 여러개 있다.

표 15-2는 STL에서 사용하는 련상용기들을 요약한다.

표 15-2. 기본련상용기

련상용기	특 성
set	오직 건객체만 보관한다. 매개 값에 단 하나의 건만 허용된다.
multiset	건객체만 보관한다. 여러개의 건값이 허용된다.
map	값객체와 건객체를 련결한다. 매개 값에는 단 하나의 건만 허용된다.
multimap	값객체와 건객체를 련결한다. 여러개의 건값이 허용된다.

련상용기의 창조는 순차용기와 같다.

```
set<int> intSet;
```

또는

```
multiset<Emplyee> machinists;
```

3) 성원함수

알고리듬은 STL의 중요한 요소로서 정렬과 검색과 같은 복잡한 조작을 수행한다. 그러나 용기는 특정형의 용기에 고유한 간단한 과제를 수행하는 성원함수들을 요구한다.

표 15-3은 자주 사용하는 용기클라스에 공통적인 성원함수들의 이름과 그 목적을 주었다.

표 15-3. 모든 용기들이 공통으로 가지는 성원함수들의 일부

이름	목적
size()	용기안의 항목수를 돌려준다.
empty()	용기가 비였으면 true를 돌려준다.
max_size()	최대로 가능한 용기의 크기를 돌려준다.
begin()	용기를 정방향으로 순환하기 위하여 반복자를 용기의 선두로 보낸다.
end()	반복자를 용기의 끝으로 보낸다. 정방향순환을 끝내는데 사용한다.
rbegin()	용기를 역방향으로 순환하기 위하여 용기의 끝으로 역반복자를 보낸다.
rend()	역반복자를 용기의 선두로 보낸다. 역방향순환을 끝내는데 사용한다.

대부분의 함수는 일정한 용기 혹은 일정한 부류의 용기에만 있다.(부록 8)

4) 용기접속기

우에서 언급한 일반용기로부터 용기접속기(adapter)라는 구조를 사용하여 특수목적용기를 창조할수 있다. 특수목적용기는 일반용기보다 더 단순한 대면부를 가진다. STL에서 용기접속기를 사용하여 실현한 특수용기들은 stack, queue, priority_queue이다. stack는 탄창꼭대기에만 자료항목을 밀어넣고 꺼낼수 있다. queue에서는 한끝으로 자료를 밀어넣고 다른 끝에서 자료를 꺼낸다. priority_queue에서는 불규칙적인 순서로 자료를 밀어넣지만 다른 끝에서 자료를 꺼낼 때 항상 보관된 가장 큰 항목을 꺼낸다. priority_queue은 자동적으로 자료를 정렬한다.

stack, queue, priority_queue는 각이한 순차용기로부터 창조할수 있다. 또한 여기에 deque를 자주 사용한다. 표 15-4에는 추상자료형과 그 실현에 사용할수 있는 순차용기를 보여준다.

표 15-4. 접속기에 기초하는 용기

용기	실현	특성
stack	vector, list, deque로부터 실현가능	삽입(넣기)과 삭제(꺼내기)를 한끝에서만 할 수 있다.

용기	실현	특성
queue	list, deque로부터 실현가능	한끝에서는 삽입(넣기)하고 다른끝에서는 삭제(꺼내기)한다.
priority_queue	vector, deque로부터 실현가능	한끝에서 분류순으로 삽입(넣기)하고 다른 끝에서 분류순으로 삭제(꺼내기)한다.

이 클라스들을 실례화하기 위하여 형판안에 형판을 사용할수 있다. 실례로 deque 클라스로부터 실례화된 int형을 보관하는 탄창객체가 있다.

```
stack< deque<int> > aStack;
```

이때 두개의 닫긴 괄호사이에는 공백을 입력해야 한다. 즉

```
stack<deque<int>> aStack;   // 오유
```

는 번역프로그람이 >>를 연산자로 해석하므로 이렇게 쓸수 없다.

2. 알고리듬

알고리듬은 용기 혹은 용기안의 항목들에 대하여 조작하는 함수이다. STL의 알고리듬은 초기의 용기서고들처럼 성원함수도 아니고 용기클라스의 동료도 아니며 독자적인 형판함수이다. 알고리듬은 기본 C++배렬과 함께 사용할수도 있고 자기가 창조한 용기클라스와 사용할수도 있다. 표 15-5는 몇가지 대표적인 알고리듬을 보여준다.(부록 8)

표 15-5 . 대표적인 STL알고리듬

알고리듬	목적
find	주어진 값과 등가한 원소를 돌려준다.
count	주어진 값을 가지는 원소들의 개수를 계수한다.
equal	두개 용기의 내용을 비교하고 대응하는 원소들이 모두 같으면 true를 돌려준다.
search	한 용기의 값들의 렬과 등가한 렬이 다른 용기안에 있는가를 검색한다.
copy	한 용기로부터 다른 용기로 값들의 렬을 복사한다.(또는 같은 용기안의 다른 위치에 복사한다.)
swap	한 위치의 값을 다른 위치의 값과 교체한다.
iter_swap	한 위치의 값들의 렬과 다른 위치의 값들의 렬을 교체한다.
fill	위치의 렬에 값을 복사한다.
sort	용기의 값들을 주어진 순서로 분류한다.
merge	두개의 정렬된 원소들을 결합하여 하나의 가장 큰 범위를 만든다.
accumulate	주어진 범위의 원소들의 합을 돌려준다.
for_each	용기안의 매개 원소에 대하여 주어진 함수를 실행한다.

자료를 가지고있는 int형의 배렬을 창조한다고 가정하자.

```
int arr[8] = { 42,31,7,80,2,26,19,75 };
```

STL sort()알고리듬에 의하여 배렬을 정렬할수 있다.

```
sort(arr, arr+8);
```

여기서 arr는 배렬의 선두주소, arr+8은 배렬의 마지막원소의 다음주소(past-the-end address)이다.

3. 반복자

반복자는 지적자와 비슷하고 용기안의 개별적자료항목(원소)을 호출하는데 쓰인다. 반복자는 원소들을 하나씩 호출하는데 사용되는데 이것을 용기를 항행하는 반복(순환)이라고 한다. ++연산자로 반복자를 증가시켜 다음 원소를 가리키게 하고 *연산자로 그것을 간접참고하여 그것이 가리키는 원소의 값을 얻을수 있다. STL에서 반복자는 iterator클라스의 객체로 표시된다.

반복자는 각이한 클라스에서 각이한 형의 용기와 함께 사용된다. 세개의 중요한 반복자클라스 즉 forward, bi_directional, random이 있다. forward반복자는 용기를 정방향으로 하나씩 항행할수 있다. ++연산자로 그것을 수행한다. forward는 뒤로 항행할수 없고 용기중간의 임의의 위치로 설정될수 없다. bi_directional반복자는 정방향은 물론 역방향으로 항행할수 있으며 ++연산자와 --연산자가 정의되여있다. random반복자는 앞뒤로 이행하는것외에 임의의 위치로 이행할수 있다. 실례로 27위치를 직접 호출할수 있다.

또한 두 종류의 특수반복자가 있다. 입력반복자는 순차자료항목들을 용기로 읽어들이기 위한 입력장치(cin 또는 파일)를 지적한다. 출력반복자는 출력장치(cout 또는 파일)를 지적하고 용기로부터 장치로 원소들을 써넣는다.

정방향, 량방향, 직접호출반복자의 값들은 정렬할수 있지만 입력과 출력반복자의 값들은 그렇게 할수 없다. 즉 처음 세개의 반복자는 기억위치를 지적하고 입력과 출력반복자는 보관된 《지적자》값이 의미를 가지지 않는 입출력장치를 가리킨다. 표 15-6은 각종 반복자들의 특성을 보여준다.

표 15-6. 반복자의 특성

반복자형	읽기와 쓰기	반복자를 보관가능	방향	호출
직접호출	읽기,쓰기	가능	정방향,역방향	직접(불규칙)
량방향	읽기,쓰기	가능	정방향,역방향	선형
정방향	읽기,쓰기	가능	정방향만	선형
출력	쓰기만	불가능	정방향만	선형
입력	읽기만	불가능	정방향만	선형

4. STL과 관련한 문제

STL형판클라스의 복잡화는 번역프로그람에 대한 긴장을 조성하고 그전보다 잘 동

작하지 않게 한다. 여기에 몇가지 문제가 있다.

우선 오유가 클라스사용자의 코드안에 있을 때, 머리부파일안에 오유가 있을 때 번역프로그람이 그것을 찾기 힘들다. 오유를 찾으려면 자기 코드의 행마다 설명문을 쓰는것과 같은 방법으로 다시 조사하여야 한다.

머리부파일의 재번역은 머리부파일을 제공하는 번역프로그람에 대하여 번역속도를 급속히 떨구고 STL과 관련한 문제를 일으킨다. 작업량이 많아보이면 사전번역된 머리부를 차단해야 한다.

STL은 그럴듯한 번역프로그람경고를 발생시킬수도 있다.

그렇지만 STL은 복잡하고 든든하고 만능적인 체계이다. 오유는 실행시보다 번역시에 포착하려고 한다. 알고리듬과 용기는 매우 안전한 대면부를 제공하고 하나의 용기나 알고리듬과 작업하는것은 다른것과도 작업한다.

제 2 절. 알고리듬

STL알고리듬은 자료집합에 대한 조작을 수행한다. 이 알고리듬은 STL용기와 작업하도록 설계되였으나 보통 C++배렬에도 적용할수 있다. 이것은 배렬프로그람작성에서 상당한 품을 줄인다. 또한 용기와 알고리듬에 대하여 학습하는 간단한 방법을 제공한다. 이 절에서는 몇가지 알고리듬을 사용하는 방법을 설명한다.

1. find()알고리듬

find()알고리듬은 용기안에서 주어진 이름을 가지는 첫 원소를 검색한다. 실례 15-1은 int배렬에서 어떤 값을 검색한다.

(실례 15-1) 주어진 값을 가진 첫 객체의 검색

```
#include <iostream>
#include <algorithm>
using namespace std;
int arr[] = { 11, 22, 33, 44, 55, 66, 77, 88 };
int main()
{
   int* ptr;
   ptr = find(arr, arr+8, 33);
   cout << "위치 " << (ptr - arr)
        << "에서 값이 33인 첫 객체를 발견하였습니다."
        << endl;
   return 0;
}
```

프로그람의 출력은 다음과 같다.

위치 2에서 값이 33인 첫 객체를 발견하였습니다.

보통 배렬의 첫 원소는 첨수가 0이므로 33은 위치 3이 아니라 2이다.

1) 머리부파일

실례에서는 머리부파일 ALGORITHM을 요구한다. 표준 C++서고안의 다른 머리부파일처럼 파일확장자가 없다. 이 파일은 STL알고리듬의 선언을 포함한다. 다른 머리부파일은 용기와 기타 목적에 사용한다. STL의 낡은 판을 사용한다면 다른 이름으로 머리부파일을 포함하여야 한다.(례를 들면 ALGO.H)

2) 범위

find()에서 처음의 두개 파라메터는 시험하려는 원소들의 범위를 지정한다. 이 값들은 반복자에 의하여 지정된다. 실례 15-1는 보통 C++지적자값을 사용하는 반복자의 확실한 경우이다.

첫 파라메터는 시험하려는 첫째 값이 반복자(이 경우에 지적자)이다. 둘째 파라메터는 시험하려는 마지막원소의 다음 위치의 반복자이다. 원소가 8개이므로 이 값은 처음값 + 8이다. 이것은 마감원소 다음값이다. 이것은 시험하려는 단위의 끝에서 하나 다음의 원소를 지적한다.

이 문법은 for순환에서 보통 C++의 성구를 련상시킨다.

```
for(int j=0; j<8; j++)      // 0-7
{
  if(arr[j] == 33)
  {
     cout << "위치 "<<j<< "에서 값 33을 가진 첫 객체를 발견하였습니다."
          <<endl;
     break;
  }
}
```

실례 15-1에서 find()알고리듬은 for순환을 쓰는데 드는 품을 줄인다. 복잡한 경우에도 알고리듬은 코드를 간단히 쓸수 있게 한다.

2. count()알고리듬

알고리듬 count()는 용기안에 주어진 값이 몇개 있는가를 계산하고 그 수를 돌려준다.

(실례 15-2) 주어진 값을 가진 객체들의 계수

```
#include <iostream>
#include <algorithm>
using namespace std;
int arr[] = { 33, 22, 33, 44, 33, 55, 66, 77 };
int main()
{
  int n = count(arr, arr+8, 33);
  cout << "배렬에 33이 " << n << "개 있습니다." << endl;
  return 0;
}
```

출력은 다음과 같다.

배렬에 33이 3개 있습니다.

3. sort()알고리듬

분류알고리듬은 sort()이다.

(실례 15-3) 옹근수배렬의 정렬

```
#include <iostream>
#include <algorithm>
using namespace std;
int arr[] = { 0, 2, -17, 22, 85, -30, 45, 25 };
int main()
{
   sort(arr, arr+8);
   for(int j=0; j<8; j++)
      cout << arr[j] << ' ';
   cout << endl;
   return 0;
}
```

프로그람의 출력은 다음과 같다.

-30, -17, 0, 2, 22, 25, 45, 85

4. search()알고리듬

일부 알고리듬은 한번에 두개의 용기에 대하여 작업한다. 례를 들면 find()알고리듬이 한개 용기에서 주어진 값을 검색한다면 search()알고리듬은 한 용기에 의해 주어진 값들의 렬을 다른 용기에서 찾는다.

(실례 15-4) 한 용기의 렬이 다른 용기에 있는가의 검색

```
#include <iostream>
#include <algorithm>
using namespace std;
int source[] = { 11, 44, 33, 11, 22, 33, 11, 22, 44 };
int pattern[] = { 11, 22, 33 };
int main()
{
   int *ptr;
   ptr = search(source, source+8, pattern, pattern+3);
   if(ptr == source+9)
      cout << "일치하는것이 없습니다.\n";
   else
      cout << (ptr - source) << "위치에서 일치하였습니다" << endl;
   return 0;
}
```

알고리듬은 배렬 source안에서 배렬 pattern에 의해 주어진 렬 11,22,33을 찾는다. 이 렬이 source의 4번째 원소(원소 3)로부터 시작되는것을 발견한다. 출력은 다음과

같다.

```
3위치에서 일치하였습니다
```

반복자값 ptr가 source의 마감원소의 다음에서 끝나면 일치하는것이 없다.

search()와 같은 알고리듬에 대한 인수들은 용기와 같은 형을 요구하지 않는다. 원천은 STL의 vector이고 견본은 배렬일수도 있다. 이와 같은 일반성은 STL의 매우 강력한 특성이다.

5. merge()알고리듬

세개의 용기와 작업하고 두개 원천용기의 원소들을 하나의 목적용기에 결합하는 알고리듬이 있다.

(실례 15-5) 두개의 용기를 셋째 용기에 결합

```
#include <iostream>
#include <algorithm>
using namespace std;
int src1[] = { 2, 3, 4, 6, 6 };
int src2[] = { 1, 3, 5 };
int dest[8];
int main()
{
   merge(src1, src1+5, src2, src2+3, dest);
   for(int j=0; j<8; j++)
      cout << dest[j] << ' ';
   cout <<endl;
   return 0;
}
```

출력은 목적용기의 내용을 표시한다.

```
1 2 3 3 4 5 6 8
```

결합은 순서를 보존하고 원천원소들의 두개 렬을 목적용기에 혼합한다.

6. 함수객체

일부 알고리듬은 인수로서 함수객체를 가진다. 함수객체는 형판함수와 비슷하다. 함수객체는 실제로 단일한 성원함수로서 재정의되는 ()연산자를 가지는 형판클라스의 객체이다.

수들의 배렬을 작아지는 순서로 구성하도록 정렬한다고 하자.

(실례 15-6) greater<>()함수객체를 사용하여 double형의 배렬을 반대순서로 정렬하기

```
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;
double fdata[] = { 19.2, 87.4, 33.6, 55.0, 11.5, 42.2 };
int main()
{
```

```
    sort(fdata, fdata + 6, greater<double>());
    for(int j=0; j<6; j++)
        cout << fdata[j] << ' ';
    cout <<endl;
    return 0;
}
```

sort()알고리듬은 보통 커지는 순서로 정렬하지만 sort()의 제3인수 greater<>()함수객체를 사용하여 정렬순서를 역전시킨다. 실행결과는 다음과 같다.

```
87.4, 55, 42.2, 33.6, 19.2, 11.5
```

이와 대조되는 변수와 론리조작용 함수객체가 있다.

1) 함수객체의 위치에 있는 사용자정의함수

함수객체는 기본 C++형들과 적당한 연산자(+, <, == 등)들이 정의된 클라스들에 대하여 동작한다. 그렇지 않은 경우에 작업하자면 사용자정의함수로 함수객체를 대신하여야 한다. 실례로 연산자 <는 보통 char*문자렬에 대하여 정의되지 않지만 그 비교함수를 쓰지 않고 함수객체의 위치에서 이 함수의 주소(함수이름)를 사용한다. 실례 15-7은 char*문자렬배렬을 정렬하는 방법을 보여준다.

(실례 15-7) 사용자정의의 비교함수객체를 사용하여 문자렬배렬을 정렬하기

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
string names[] = { "Kim", "Li", "Pak", "Cha", "O", "SonU" };
bool AlphaComp(char*, char*);
int main()
{
    sort(names, names + 6, AlphaComp);
    for(int j=0; j<6; j++)
        cout << names[j] << endl;
    cout <<endl;
    return 0;
}
bool AlphaComp(char* s1, char* s2)
{
    return (strcmp(s1, s2) < 0) ? true : false;
}
```

sort()알고리듬의 제3인수는 AlphaComp()함수의 구조이다. AlphaComp()함수는 두개의 char*문자렬을 비교하고 첫째가 둘째보다 자모순으로 작은가에 따라서 true 혹은 false를 돌려준다. 그것은 제1인수가 제2인수보다 작으면 0아닌 값을 돌려주는 C서고함수 strcmp()를 사용한다. 이 프로그람의 출력은 기대한것과 같다.

```
Cha
Kim
Li
O
```

```
Pak
SonU
```

실제로 본문을 조작하기 위하여 자기의 함수객체를 쓸 필요는 없다. 표준서고로부터 string클라스를 사용하면 less<>()와 greater<>()와 같은 내부함수객체를 사용할수 있다.

2) 알고리듬에 _if의 추가

일부 알고리듬은 _if로 끝나는 판을 가진다. 이 알고리듬은 술어라고 부르는 여유 파라메터를 가진다. 술어(predicate)는 함수객체 혹은 함수이다. 실례로 find()알고리듬은 주어진 값과 같은 원소들을 모두 찾는다. 또한 find_if()알고리듬과 작업하고 임의의 특성을 가진 원소들을 탐색하는 함수를 창조할수 있다.

다음의 실례는 string객체를 사용한다. find_if()알고리듬은 사용자가 작성한 IsPak()함수를 인수로 하여 호출되고 string객체배렬에서 "Pak"값을 가지는 첫 string을 검색한다.

(실례 15-8) "Pak"이라는 이름을 가진 문자렬의 탐색

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
bool IsPak(string name)
{
  return name == "Pak";
}
string names[] = { "Kim", "Li", "Pak", "Cha", "SonU" };
int main()
{
  string* ptr;
  ptr = find_if(names, names + 5, IsPak);
  if(ptr == names + 5)
     cout << "Pak은 이 목록에 없다.\n");
  else
     cout << "Pak은 목록의 " << (ptr - name) << "원소이다.\n";
  return 0;
}
```

"Pak"은 배렬안의 이름들중의 하나이고 프로그람의 출력은 다음과 같다.

```
Pak은 목록의 2원소이다.
```

함수 IsPak()의 정의는 find_if()의 제3인수이고 첫째와 둘째 인수는 보통 배렬의 첫 원소와 마지막 원소의 다음주소이다.

find_if()알고리듬은 배렬의 매개 원소에 IsPak()함수를 적용한다. IsPak()은 임의의 원소에 대하여 bool값을 돌려주고 find_if()는 그 원소의 지적자(반복자)의 값을 돌려준다. 다른 한편 배렬의 마지막 원소의 다음주소를 가지는 지적자를 돌려준다.

count(), replace(), remove()와 같은 알고리듬은 _if판을 가진다.

7. for_each()알고리듬

for_each()알고리듬은 용기안의 매개 항목에 어떤 조작을 하게 한다. 작성자의 함수는 용기안의 원소들을 변경할수 없으나 그 값을 사용하여 표시할수 있다.

여기에 for_each()가 인치로부터 센치메터로 배렬의 모든 값들을 변환하고 그 값들을 표시하는데 사용하는 실례가 있다. 2.54를 값에 곱하는 InToCm()이라는 함수를 쓰고 이 함수의 주소를 for_each()의 제3인수로 사용한다.

(실례 15-9) for_each()의 사용

```
#include <iostream>
#include <algorithm>
using namespace std;
void InToCm(double);
int main()
{
    double inches[] = { 3.5, 6.2, 1.0, 12.71, 4.33 };
    for_each(inches, inches + 5, InToCm);
    cout <<endl;
    return 0;
}
void InToCm(double in)
{
    cout << (in * 2.54) << ' ';
}
```

출력은 다음과 같다.

```
8.89 15.748 2.54 31.385 10.9982
```

8. transform()알고리듬

transform()알고리듬은 용기안의 매개 항목에 어떤 조작을 하고 다른 용기(또는 같은 용기)에 결과값들을 보관한다. 또한 사용자가 쓴 함수는 매개 항목에 대하여 수행하는 동작을 결정한다. 함수의 돌림값은 목적용기와 같아야 한다. 우의 실례는 실례 15-9와 비슷하다. 그러나 InToCm()함수는 변환된 값들을 표시하지 않고 다른 배렬 centi[]에 센치메터의 값으로 보관한다. 그다음 프로그람은 centi[]의 내용을 표시한다.

(실례 15-10) transform()알고리듬

```
#include <iostream>
#include <algorithm>
using namespace std;
double InToCm(double);
int main()
{
    double inches[] = { 3.5, 6.2, 1.0, 12.71, 4.33 };
    double centi[5];
    transform(inches, inches + 5, centi, InToCm);
```

```
    for(int j=0; j<5; j++)
        cout << centi[j] << ' ';
    cout <<endl;
    return 0;
}
double InToCm(double in)
{
    return (in * 2.54);
}
```
출력은 실례 15-9의 형식과 같다.

제 3 절. 순차용기

이미 언급한것처럼 STL에는 두 종류의 용기 즉 순차용기와 련상용기가 있다. 이 절에서는 세개의 순차용기 vector, list, deque의 동작방법과 성원함수들을 고찰한다. 반복자를 아직 학습하지 않았으므로 이 용기에 대하여 수행할수 없는 일부 조작이 있다.

매개 실례에서는 용기의 여러 성원함수들을 소개한다. 다른 종류의 용기들도 같은 이름과 특성을 가지고 성원함수를 사용한다. 실례로 push_back()는 목록과 대기렬과 련관될수도 있다.

1. vector

벡토르를 고도의 배렬로 생각할수 있다. 벡토르는 자료를 삽입하거나 삭제할 때 크기를 증가시킴으로써 기억할당을 관리할수 있다. 벡토르에서도 배렬처럼 []연산자로 원소를 호출할수 있다. 이러한 직접호출은 벡토르에서 매우 고속이다. 또한 벡토르의 끝에 새로운 자료항목을 추가(밀어넣기)하는 조작이 매우 고속이다. 이때 벡토르의 크기는 새로운 항목을 유지할수 있도록 자동적으로 증가된다.

1) 성원함수 push_back(), size(), operator[]

실례 15-11은 가장 일반적인 벡토르조작이다.

(실례 15-11) 벡토르의 성원함수 push_back(), operator(), size()
```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v;
    v.push_back(10);
    v.push_back(11);
    v.push_back(12);
    v.push_back(13);
```

```
    v[0] = 20;
    v[3] = 23;
    for(int j=0; j<v.size(); j++)
       cout << v[j] << ' ';
    cout <<endl;
    return 0;
  }
```

vector의 기본구성자에 의하여 벡토르 v를 창조한다. 모든 STL용기들처럼 용기가 보관하는 변수의 형을 지정하는데 형판형식을 사용한다. 이 경우에 int형용기의 크기는 지적하지 않으므로 0으로 시작한다.

push_back()성원함수는 벡토르의 뒤에 인수값을 삽입한다.(여기서는 뒤에 추가되는 원소의 첨수가 최대로 된다.) 벡토르의 앞(첨수 0인 원소)에는 목록이나 대기렬과 달리 새로운 원소를 삽입할수 없다. 여기서는 값 10, 11, 12, 13을 뒤에 밀어넣는다. v[0]은 10, v[1]은 11, v[2]은 12, v[3]은 13을 포함한다.

일단 벡토르가 자료를 가지면 그 자료는 재정의된 []연산자에 의하여 마치도 배렬처럼 호출할수(읽고 쓸수) 있다. []연산자로 첫째 원소 10을 20으로, 마지막 원소 13을 23으로 변경한다. 출력은 다음과 같다.

```
20 11 12 23
```

size()성원함수는 현재 용기안의 원소수를 돌려준다. 실례 15-11에서 그것은 4이다. for순환에서는 이 값에 기초하여 용기안의 원소값들을 모두 출력한다.

다른 성원함수 max_size()는 용기를 확장할수 있는 최대크기를 돌려준다. 이 수는 용기에 보관하는 자료형(원소가 클수록 보관할수 있는 개수는 적다.), 용기의 형, 조작체계에 의존한다. 실례로 32bit체계에서 int형벡토르에 대하여 max_size()는 1,073,741,823이다.

2) 성원함수 swap(), empty(), back(), pop_back()

실례 15-12는 추가적인 벡토르구성자들과 성원함수들을 보여준다.

(실례 15-12) 벡토르의 구성자, swap(), empty(), back(), pop_back()

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
  double arr[] = { 1.1, 2.2, 3.3, 4.4 };
  vector<double> v1(arr, arr+4);
  vector<double> v2(4);
  v1.swap(v2);
  while(!v2.empty())
  {
     cout << v2.back() << ' ';
     v2.pop_back();
  }
```

```
    cout <<endl;
    return 0;
  }
```

실례에서는 두개의 새로운 벡토르구성자를 사용한다. 첫째는 벡토르 v1을 인수로 넘어온 보통의 C++배렬의 값으로 초기화한다. 이 구성자에 대한 인수는 배렬의 선두에로의 지적자와 마지막 원소의 다음에로의 지적자이다. 둘째 구성자는 v2을 초기크기 4로 설정하지만 초기값을 주지 않는다. 두 벡토르는 double형이다.

swap()성원함수는 한 벡토르의 모든 자료를 다른 벡토르의 자료들과 순서를 유지하면서 교체한다. 이 프로그람에서는 v2에 쓸모없는 자료가 있으므로 v1의 자료와 교체한다. v1에 있는 자료의 내용을 보기 위하여 v2을 표시한다. 출력은 다음과 같다.

```
4.4,3.3,2.2,1.1
```

back()성원함수는 벡토르의 마지막 원소의 값을 돌려준다. 그 값을 cout로 표시한다. pop_back()성원함수는 벡토르에서 마지막 원소를 삭제한다. 따라서 순환할 때마다 다른 마지막 원소가 있게 된다. pop_back()는 탄창의 pop()처럼 마지막 원소의 값을 돌려주고 벡토르에서 그것을 삭제하지만 back()는 그것을 삭제하지 않는다.

swap()와 같은 일부 성원함수는 알고리듬처럼 존재한다. 이때 성원함수판은 보통 알고리듬판보다 특정한 용기에서 보다 효과있다. 때로는 알고리듬을 사용할수도 있다. 실례로 두개의 다른 종류의 용기에 있는 원소들을 바꾸는데 사용할수 있다.

3) 성원함수 insert()와 erase()

insert()와 erase()성원함수는 용기안의 임의의 위치에 원소를 삽입하거나 삭제한다. 이 함수들은 벡토르에 매우 효과있다. 삽입후에는 새로운 원소용공간을 만들기 위하여, 혹은 삭제된 항목이 있던 공간을 없애기 위하여 모든 원소를 이동하여야 한다. 그러나 삽입과 삭제는 그 속도가 지수적이면 사용할수 없다. 실례 15-13은 이 함수를 보여준다.

(실례 15-13) 벡토르의 insert(), erase()

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
  int arr[] = { 100, 110, 120, 130 };
  vector<int> v(arr, arr+4);
  cout << "\n삽입전:";
  for(int j=0; j<v.size(); j++)
     cout << v[j] << ' ';
  v.insert(v.begin() + 2, 115);
  cout << "\n삽입후:";
  for(j=0; j<v.size(); j++)
     cout << v[j] << ' ';
  v.erase(v.begin() + 2);
```

```
    cout << "\n삭제후:";
    for(j=0; j<v.size(); j++)
        cout << v[j] << ' ';
    cout <<endl;
    return 0;
}
```

insert()성원함수는 두개의 인수 즉 원소를 용기안에 삽입하는 위치와 원소의 값을 가진다. 벡토르에서 원소 2(세번째 원소)를 지정하기 위하여 begin()원소에 2를 더한다. 삽입점으로부터 용기끝까지의 원소들을 우로 이동하여 빈자리를 만드므로 용기의 크기는 하나 증가한다.

erase()성원함수는 특정위치에 있는 원소를 삭제한다. 삭제점우의 원소들은 아래로 이동되고 용기의 크기는 하나 감소된다. 실례 15-13의 출력은 다음과 같다.

```
삽입전: 100 110 120 130
삽입후: 100 110 115 120 130
삭제후: 100 110 120 130
```

2. list

STL목록의 용기는 2중련결목록이다. 목록의 매개 원소는 다음 원소에로의 지적자뿐아니라 앞원소에로의 지적자도 포함한다. 용기는 앞(처음)과 뒤(마지막) 원소들의 주소를 둘다 보관하고 이것은 목록의 량끝에로의 고속호출을 가능하게 한다.

1) 성원함수 push_front()와 front(), pop_front()

실례 15-14는 자료들을 앞뒤에 밀어넣고 읽고 꺼내는 방법을 보여준다.

(실례 15-14) 목록의 push_front(), front(), pop_front()

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
    list<int> iList;
    iList.push_back(30);
    iList.push_back(40);
    iList.push_front(20);
    iList.push_front(10);
    int size = iList.size();
    for(int j=0; j<iList.size(); j++)
    {
        cout << iList.front() << ' ';
        iList.pop_front();
    }
    cout <<endl;
    return 0;
}
```

목록의 앞뒤에 자료를 밀어넣고 표시하며 보통순서로 앞에서부터 자료를 삭제한다.

```
10 20 30 40
```

push_front()와 pop_front(), front()성원함수는 벡토르에서 본 push_back()와 pop_back(), back()와 비슷하다. 목록의 원소들에 대해서는 속도가 대단히 느리므로 직접호출을 사용할수 없다. 따라서 목록에는 []연산자가 정의되지 않는다. 있다면 이 연산자는 목록을 순환하면서 원소들을 계수하여 정확한 원소로 이동해가는 시간을 소비하는 조작을 한다. 직접호출이 필요하면 vector나 deque를 사용해야 한다.

목록은 그 중간에서 삽입과 변경을 자주 진행할 때 적합하다. 이것은 vector와 deque에 비효과적이다. 그것은 삽입 혹은 삭제된 원소 다음의 모든 원소들을 이동해야 하기때문이다. 그러나 목록에서는 몇개의 지적자만 있으면 매개 항목을 삽입하거나 삭제하기 위한 변경이 가능하고 고속이다.(그러나 정확한 삽입점을 찾는데 여전히 시간을 소비한다.)

insert()와 erase()성원함수는 목록에로의 삽입과 삭제에 사용되지만 반복자의 사용을 요구하지 않으므로 이 함수들의 설명을 뒤로 미룬다.

2) 성원함수 reverse()와 merge(), unique()

일부 성원함수는 목록에만 존재한다. 즉 같은 일을 하는 알고리듬은 있지만 다른 용기에는 없다. 실례 15-15는 이 함수를 보여준다. 두개 배렬의 내용으로 두개의 int 목록의 객체를 채우는것으로 시작한다.

(실례 15-15) 목록의 reverse(), merge(), unique()

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
   int j;
   list<int> list1, list2;
   int arr1[] = { 40, 30, 20, 10 };
   int arr2[] = { 15, 20, 25, 30, 35 };
   for(j=0; j<4; j++)
      list1.push_back(arr1[j]);
   for(j=0; j<5; j++)
      list2.push_back(arr2[j]);
   list1.reverse();
   list1.merge(list2);
   list1.unique();
   int size = list1.size();
   while(!list1.empty())
   {
      cout << list1.front() << ' ';
      list1.pop_front();
   }
   cout <<endl;
   return 0;
}
```

첫 목록은 역순으로 되여있으므로 reverse()성원함수를 사용하여 보통의 정렬순서로 변환한다.(량끝을 호출할수 있으므로 목록용기의 역전은 고속이다.)

셋째 성원함수 merge()는 두개 목록에 조작하여 그것들을 정렬된 순서로 만든다.

10,20,30,40

15,20,25,30,35

그러면 merge()함수는 매개의 정렬순서와 모든 원소들을 보관하도록 list1을 확장하면서 list2을 list1에 결합한다. list1의 결과는 다음과 같다.

10,15,20,20,25,30,30,35,40

끝으로 list1에 unique()성원함수를 적용한다. 이 함수는 같은 값을 가지는 이웃원소들을 찾고 처음것을 제외하고는 모두 삭제한다. 그리고 list1의 내용을 다시 표시한다. 즉

10,15,20,25,30,35,40

목록의 내용을 표시하기 위하여 for순환에서 front()와 pop_front()성원함수를 사용한다. 선두로부터 마지막까지의 매개 원소를 표시하고 목록에서 꺼낸다. 목록을 표시하는 과정이 곧 그것을 해체하는 과정이다. 이것은 우리가 늘 요구하는것이 아니지만 지금까지 설명한 련속적인 목록원소들을 호출하는 유일한 수법이다.(반복자에 의하여 이 문제를 해결한다.)

3. deque

deque의 일부는 vector와 비슷하고 일부는 list와 비슷하다. vector처럼 deque는 []연산자를 사용한 직접호출을 할수 있다. 그러나 list처럼 deque는 뒤는 물론 앞으로도 호출할수 있다. 이것은 push_front()와 pop_front(), front()를 가지는 량끝 벡토르의 부류이다.

기억기는 벡토르나 대기렬에 대하여 서로 다르게 할당된다. vector는 항상 기억기의 련속령역을 차지한다. 벡토르가 너무 커지면 적당한 새로운 위치로 이동하여야 한다.

다른 한편 deque는 여러개의 비련속령역에 보관되고 토막화된다. 성원함수 capacity()는 벡토르를 이동하지 않고 보관할수 있는 최대원소수를 돌려준다. 그러나 deque에서는 원소들을 옮길 필요가 없으므로 capacity()가 정의되지 않는다.

(실례 15-16) deque의 push_back(), push_front(), front()

```
#include <iostream>
#include <deque>
using namespace std;
int main()
{
  deque<int> deq;
  deq.push_back(30);
  deq.push_back(40);
  deq.push_back(50);
  deq.push_front(20);
  deq.push_front(10);
  deq[2] = 33;
  for(int j=0; j<deq.size(); j++)
     cout << deq[j] << ' ';
  cout <<endl;
  return 0;
```

}

이미 push_back()와 push_front(), operator[]의 실례를 보았다. 이 함수들은 다른 용기들에서도 deque와 꼭같이 작업한다. 이 프로그람의 출력은 다음과 같다.

10 20 30 40 50

그림 15-2는 세가지 순차용기의 중요한 성원함수들을 보여준다.

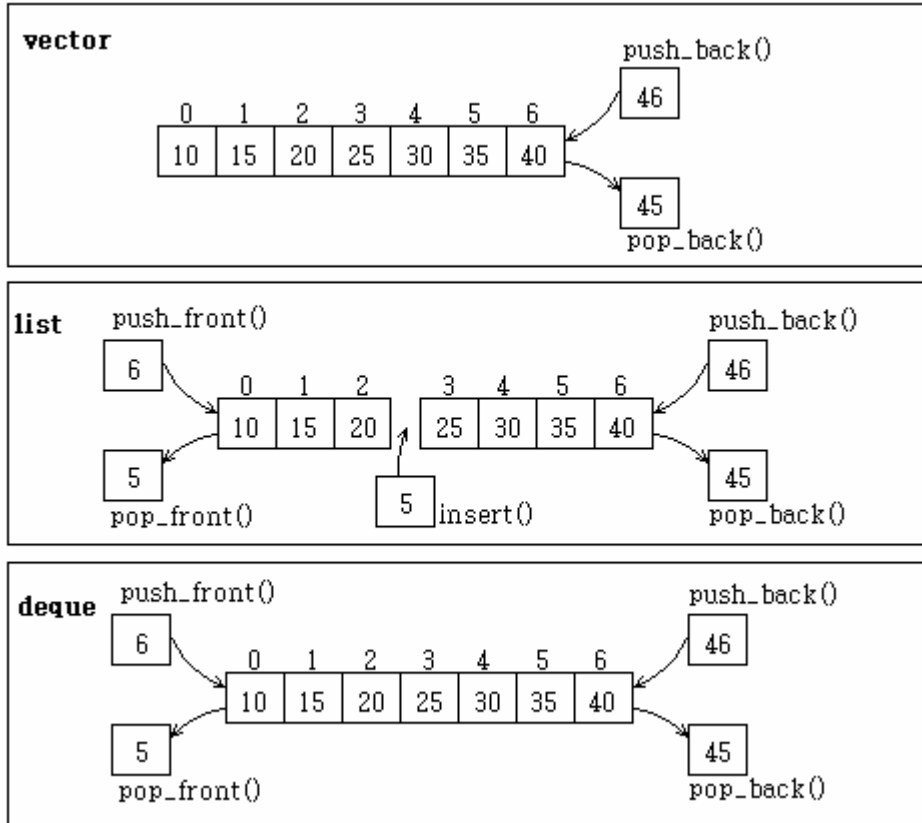



그림 15-2. 순차용기

제 4 절. 반복자

반복자는 좀 신비한것 같지만 아직 STL의 조작에서 중심으로 되고있다. 이 절에서는 우선 반복자가 수행하는 두가지 역할 즉 고도의 지적자로서, 그리고 알고리듬과 용기사이의 대면부로서의 역할을 설명한다. 그다음 그 사용실례를 보여준다.

1. 고도의 지적자로서 반복자

때로는 용기안의 모든 원소(또는 원소들의 일정한 범위)에 대하여 어떤 조작을 수행하여야 한다. 용기의 매개 원소의 값을 표시하거나 값을 모두 더하는것이 그 실례이다. 보통 C++배렬에서 이러한 조작은 지적자(또는 []연산자)에 의하여 수행된다. 실례로 다음의 코드는 류동소수점수배렬을 반복하면서 매개 원소의 값을 표시한다.

```
float* ptr = start_address;
for(int j=0; j<SIZE; j++)
```

```
cout << *ptr++;
```

우리는 *연산자에 의하여 지적자 ptr를 간접참고하여 그것이 지적하는 값을 얻으며 ++연산자로 그것을 증가시켜 다음 항목을 가리키게 한다.

1) 보통지적자에는 능력이 부족하다

복잡한 용기들에서 일반의 C++지적자에는 결함이 있다. 그 하나로서 용기에 보관된 항목들이 기억기에 련속보관되지 않는다면 지적자조종은 매우 복잡해지는것이다. 따라서 다른 값을 가리키도록 지적자를 단순히 증가시킬수 없다. 실제로 련결목록에서 다음 항목에로의 이동중에서 그 항목이 이전 항목과 이웃이라고 가정할수 없고 지적자들의 사슬을 추적해야 한다.

또한 지적자변수에 어떤 용기원소의 주소를 보관하고 앞으로 어떤 시각에 그 원소를 호출하려고 한다.

용기의 중간에 무엇인가 삽입하거나 삭제하면 보관된 지적자값에 무슨 일이 생기는가?

용기의 내용이 재배렬된다면 그것은 계속 유효할수 없다. 삽입과 삭제후에 보관된 모든 지적자값들을 다시 호출하는데 대하여 근심하지 않으면 좋다.

이러한 문제에 대한 하나의 해결책은 《고도의 지적자》의 클라스를 창조하는것이다. 그러한 클라스의 객체는 그 성원함수들을 일반지적자준위에서 기본적으로 포함한다. ++와 *연산자는 지어 원소들이 기억기에 련속 배치되지 않거나 그 위치를 변경시켜도 용기안의 원소에 대하여 조작하는 방법을 알수 있게 재정의된다. 여기에 골격프로그람이 있다.

```
class SmartPointer
{
private:
   float* p;   // 보통지적자
public:
   float operatotor*()
      {}
   float operator++()
      {}
};
void main()
{
   ...
   SmartPointer sptr = start_address;
   for(int j=0; j<SIZE; j++)
      Cout << *sptr++;
}
```

2) 고도의 지적자를 어떻게 만들어야 하는가?

SmartPointer클라스를 용기에 포함해야 하는가, 또는 개별적인 클라스이여야 하는가?

STL에서 선택한 수법은 독립적인 클라스(실례로 형판화된 클라스계렬)로 준비된 지적자를 만드는것이며 이것을 반복자라고 한다. 클라스사용자는 반복자를 그러한 클라스의 객체로 정의하여 창조한다.

2. 대면부로서 반복자

반복자는 용기들에서 항목들에로의 고도의 지적자로서 동작하는것외에 STL에서의 다른 중요한 목적을 위하여 봉사한다. 반복자는 어느 알고리듬을 어느 용기에 사용할수 있는가를 결정한다. 이것이 왜 필요한가?

기본적으로 매개 용기에 매개 알고리듬을 적용할수 있어야 한다. 사실상 많은 알고리듬이 모든 STL용기와 작업하지만 일부 알고리듬은 일부 용기에서 사용할 때 매우 비효과적이다.(매우 느리다.) 실례로 sort()알고리듬은 정렬할 때 용기에로의 직접호출을 요구하는 한편 매개 원소를 이동하기 전에 탐색을 위하여 용기를 순환해야 하는데 이것은 시간을 소비한다. 마찬가지로 reverse()알고리듬은 효과성을 위하여 용기의 정방향반복과 역방향반복을 요구한다.

반복자는 용기에 적당한 알고리듬을 맞물리기 위한 아주 우수한 방법을 제공한다. 즉 콤퓨터와 인쇄기를 련결하는데 케블을 사용하는것과 같이 케블의 한끝은 용기에 꽂고 다른 끝은 알고리듬에 꽂는다. 그러나 모든 케블을 모든 용기에 꽂을수 있는것은 아니고 모든 케블을 모든 알고리듬에 꽂을수 있는것도 아니다. 주어진 용기형에 대하여 매우 강력한 알고리듬을 사용하려고 하는데 그것들을 련결하기 위한 케블(반복자)이 없다. 그런데 이 알고리듬을 사용하려고 시도하면 번역프로그람오유를 얻는다.

그러면 용기와 알고리듬을 련결하는 반복자(케블)가 몇개 있어야 하는가?

그림 15-3은 그 5종류를 보여주며 입력과 출력의 복잡성이 동일한것을 제외하고는 복잡성이 커지는 순서로 아래로부터 우로 배렬하였다.(이것은 계승도가 아니다.)

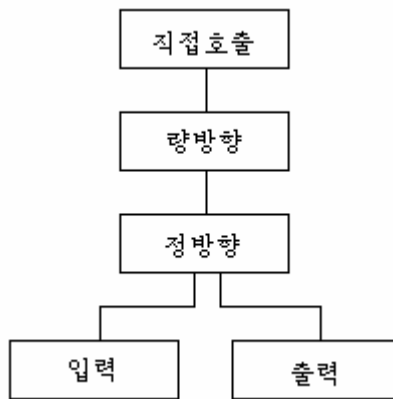



그림 15-3. 반복자종류

어떤 알고리듬이 용기를 통하여 정방향으로만 한개 항목씩 읽어야 한다면 입력(input)반복자를 사용하여 파일이름을 용기와 련결할수 있다. 실제로 입력반복자는 대체로 용기없이 파일과 cin으로부터 읽어들이는데 사용된다.

알고리듬이 용기를 통하여 정방향으로 한걸음씩 나가지만 용기로부터 읽을 대신 용기에 써넣는다면 그 반복자를 출력(output)반복자라고 한다. 출력반복자는 대체로 파일이나 cout에 써넣을 때 사용된다.

알고리듬이 용기를 정방향으로 한걸음씩 이동하면서 용기에 읽거나 써넣어야 한다면 정방향(forward)반복자를 사용해야 한다.

알고리듬이 용기를 정방향과 역방향으로 한걸음씩 이동해야 한다면 량방향(bidirectoinal)반복자를 사용한다.

끝으로 알고리듬이 용기를 차례로가 아니라 경우에 따라서 호출해야 한다면 직접호출(random)반복자를 사용해야 한다. 직접호출반복자는 배렬과 비슷하고 임의의 원소를 호출할수 있다. 이 반복자는 다음과 같이

```
iter2 = iter1 + 7;
```

산수연산자에 의하여 조작할수 있는 유일한 반복자이다. 알수 있는것처럼 모든 반복자는 용기를 정방향으로 한걸음씩 순환하는데 ++연산자를 사용한다. 입력반복자는 갈기기호의 오른변에만 *연산자를 사용할수 있다.

```
value = *iter;
```

표 15-7. 각이한 반복자범주의 능력

반복자형	한걸음전진 ++	읽기 value=*i	쓰기 *i=value	한걸음후진 --	직접호출 [n]
직접호출	○	○	○	○	○
량방향	○	○	○	○	×
정방향	○	○	○	×	×
출력	○	×	○	×	×
입력	○	○	×	×	×

출력반복자는 오른변에만 *연산자를 사용할수 있다.

```
*iter = value;
```

정방향반복자는 읽기와 쓰기를 다 조종하고 량방향반복자는 증가, 감소될수 있다. 직접호출반복자는 []연산자에 의하여 임의의 원소를 고속호출할수 있다. (물론 +, -와 같은 산수연산이 가능하다.)

알고리듬은 반복자를 그것이 요구하는 능력이상으로 사용할수 없다. 실제로 정방향반복자를 요구한다면 그것을 량방향반복자 또는 직접호출반복자에 꽂는것이 좋다.

3. 알고리듬과 용기의 련결

반복자는 알고리듬과 용기를 접속하므로 반복자와 비슷한것으로서 케블을 사용한다. 이 추상케블의 량끝(즉 용기끝과 알고리듬끝)에 초점을 두자.

1) 용기에 케블의 련결

기본STL용기로 제한한다면 오직 두 종류의 반복자를 사용할수 있다. 표 15-8과 같이 vector와 deque는 임의의 종류의 반복자를 받아들이며 list, set, multiset, multimap는 직접반복자를 제외한 어느것이나 받아들인다.

표 15-8. 용기에 받아들인 반복자형

반복자형	vector	list	deque	set	multiset	map	multimap
직접호출	○	×	○	×	×	×	×
량방향	○	○	○	○	○	○	○
정방향	○	○	○	○	○	○	○
입력	○	○	○	○	○	○	○
출력	○	○	○	○	○	○	○

그러면 STL은 주어진 용기에 알맞는 정확한 반복자의 사용을 어떻게 지적하는가? 반복자를 정의할 때 그것이 사용해야 하는 반복자의 종류를 지적해야 한다. 실례로 int형의 원소를 보관하는 목록을 정의하려면 다음과 같이 선언한다.

```
list<int> iList;    // int목록
```

그다음 목록에로의 반복자를 정의하려면 다음과 같이 선언한다.

```
list<int>::iterator iter;    // int목록에로의 반복자
```

여기서 STL은 목록을 요구하므로 반복자를 자동적으로 량방향반복자로 만든다. vector 혹은 deque에로의 반복자는 직접호출반복자로서 자동적으로 창조된다. 이 자동선택처리는 주어진 용기에 알맞는 일반용반복자클라스로부터 주어진 용기의 반복자클라스를 파생시킴으로써 실현한다.

이리하여 vector와 deque에서 반복자는 random_access_iterator클라스로부터 파생되고 목록에로의 반복자는 bidirectional_iterator클라스로부터 파생된다.

우리는 용기를 반복자(케블)의 끝에 꽂는 방법을 보았다. 케블은 용기에 실지로 꽂지 않는다. 케블은 용기에 배선된다. vector와 deque는 항상 직접호출케블과 결합되고 list(모든 련상목록)는 항상 량방향케블과 결합된다.

2) 알고리듬에 케블꽂기

앞에서 반복자케블의 한끝을 용기와 접속하는 방법을 보았다. 이로부터 케블의 다른 끝을 고찰할 준비가 되였다.

그러면 반복자를 알고리듬에 어떻게 꽂겠는가?

매개 알고리듬은 용기안의 원소들에 어떻게 작용하는가에 따라 일정한 종류의 반복자를 요구한다. 알고리듬이 용기안의 임의의 위치의 원소들을 호출해야 한다면 직접호출반복자를 요구한다. 반복자를 통하여 한걸음씩 전진하면 능력이 약한 정방향반복자를 사용할수 있다. 표 15-9는 알고리듬의 견본과 그것이 요구하는 반복자를 보여준다.(부록 7)

표 15-9. 알고리듬이 요구하는 반복자형

	입력	출력	정방향	량방향	직접호출
for_each	○				
find	○				
count	○				
copy	○	○			
replace			○		
unique			○		
reverse				○	
sort					○
nth_element					○
merge	○				
accumulate	○				

또한 매개 알고리듬이 일정한 준위의 용량을 가진 반복자를 요구할수 있지만 더 강력한 반복자가 작업한다. replace()알고리듬은 정방향반복자를 요구하지만 량방향 또는 직접호출반복자와도 작업할수 있다.

알고리듬이 콤퓨터의 케블접속구와 같이 정확한 판을 가진 접속구를 가진다고 가정하자. 이것을 그림 15-4에 보여준다. 직접호출반복자를 요구하는 알고리듬은 5개핀을 가지고 량방향반복자를 요구하는 알고리듬은 4개핀을 가지며 정방향반복자를 요구하는 알고리듬은 3개핀을 가진다.

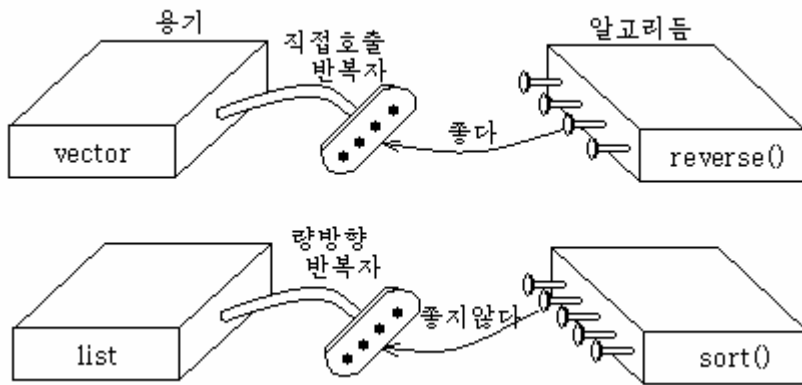



그림 15-4. 용기와 알고리듬을 련결하는 반복자

반복자(케블)의 알고리듬측은 일정한 개수의 구멍을 가진 접속구를 가진다. 5핀알고리듬에 5구멍반복자를 꽂을수 있으며 4개미만의 핀을 가진 알고리듬에도 그것을 꽂을수 있다. 그러나 5핀(직접호출)알고리듬에 4구멍(량방향)반복자를 꽂을수 없다. 따라서 직접호출반복자를 가진 vector와 deque는 알고리듬에 꽂을수 있다. 한편 목록과 련상용기는 4구멍량방향반복자를 가지고있으므로 능력이 약한 알고리듬에만 꽂을수 있다.

3) 표에 대한 고찰

표 15-8과 표 15-9는 알고리듬이 어떤 용기와 작업하는가를 보여준다. 표 15-9는 sort()알고리듬이 직접호출반복자를 요구한다는것을 보여준다. 표 15-8은 직접호출반복자를 조종할수 있는 용기들은 vector와 deque라는것을 보여준다. 물론 set, map 등에 sort()알고리듬을 적용하는 경우는 없다.

직접호출반복자를 요구하지 않는 알고리듬은 임의의 종류의 STL용기와 작업한다. 그것은 이 용기들이 모두 량방향반복자를 가지기때문이다.(STL에 단일련결목록이 있으면 그것은 정방향반복자만 사용할수 있으므로 reverse()알고리듬을 사용할수 없다.)

비교적 적은 알고리듬이 직접호출반복자를 요구하고 대다수 알고리듬은 대부분의 용기와 작업한다.

4) 성원함수와 알고리듬의 선택

보통 성원함수를 사용하겠는가 혹는 같은 이름의 알고리듬을 사용하겠는가 하는것을 선택하여야 한다. 실례로 find()알고리듬은 입력반복자만 요구하므로 임의의 용기에서 사용할수 있다. 그러나 set와 map는 순차용기와 달리 자체의 find()성원함수를 가진다.

그러면 어느 판의 find()를 사용하겠는가?

일반적으로 성원함수판이 존재하는것은 알고리듬판이 그 용기에 대하여 효과가 적기때문이므로 이 경우에 성원함수판을 사용해야 한다.

4. 반복자의 작업

반복자의 사용법은 그것에 대하여 말하는것보다 훨씬 더 간단하다. 우리는 이미 한개이상의 실례를 보았으며 거기서 반복자값들은 용기의 begin()과 end()성원함수에 의해 귀환된다. 이 성원함수들은 반복자를 지적자처럼 취급하고 반복자값을 돌려준다.

그러면 실제반복자를 그 성원함수나 기타 함수들에서 어떻게 사용하는가를 고찰하자.

1) 자료호출

직접호출반복자를 제공하는 용기(vector와 deque)중에서는 []연산자에 의하여 용기를 순환하기 쉽다. list와 같은 용기는 직접호출을 할수 없고 다른 방법을 요구한다. 이전의 실례에서는 목록의 내용을 한개 항목씩 꺼내서 표시하였다.(실례 15-14와 실례 15-15) 이것보다 더 실천적인 수법은 용기용의 반복자를 정의하는것이다. 실례 15-17이 이것을 보여준다.

(실례 15-17) 반복자와 출력용 for순환

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
int main()
{
   int arr[] = { 2, 4, 6, 8 };
   list<int> theList;
   for(int k=0; k<4; k++)
      theList.push_back(arr[k]);
   list<int>::iterator iter;
   for(iter=theList.begin() ; iter != theList.end(); iter++)
      cout << *iter << ' ';
   cout <<endl;
   return 0;
}
```

프로그람은 단순히 theList용기의 내용을 표시한다. 출력은 다음과 같다.

```
2 4 6 8
```

용기형과 일치하게 list<int>형의 반복자를 정의한다. 지적자변수처럼 반복자를 사용하기 전에 거기에 값을 주어야 한다. for순환에서는 반복자를 용기의 선두인 iList.begin()으로 초기화한다. ++연산자를 증가시켜 용기안의 원소들을 하나씩 순환한다. 또한 *연산자로 반복자를 비참고하여 그것이 가리키는 매개 원소의 값을 얻는다. 또한 !=연산자에 의하여 비교하면서 용기의 끝에 이르면 즉 iList.end()이면 순환을 끝낸다. for순환대신 while순환을 사용하는것과 같은 수법은 다음과 같다.

```
iter = iList.begin()
while(iter != iList.end())
     cout << *iter++ << ' ';
```

*iter문법은 지적자의 경우와 같다.

2) 자료의 삽입

실례 15-18과 같이 용기안의 현존원소들에 자료를 보관하는데 류사한 코드를 사용할수 있다.

(실례 15-18) 반복자를 사용하여 목록에 자료채우기

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
```

```
    list<int> iList(5);
    list<int>::iterator it;
    int data = 0;
    for(it=iList.begin(); it != iList.end(); it++) *it = data += 2;
    for(it=iList.begin(); it != iList.end(); it++) cout << *it << ' ';
    cout <<endl;
    return 0;
}
```

첫 순환은 용기를 int값 2, 4, 6, 8, 10으로 채우고 재정의된 *연산자는 갈기기호의 왼변과 오른변에 대하여 작업한다. 그리고 순환에서 이 값들을 표시한다.

3) 알고리듬과 반복자

우리가 취급한 알고리듬은 반복자를 인수(때로는 돌림값)로 사용한다. 실례 15-19는 목록에 적용한 find()알고리듬을 보여준다. 여기서 find()알고리듬을 목록에 사용할수 있다는것을 알수 있다.

(실례 15-19) 목록반복자를 돌려주는 find()

```
#include <iostream>
#include <algorithm>
#include <list>
using namespace std;
int main()
{
    list<int> theList(5);
    list<int>::iterator iter;
    int data = 0;
    for(iter=theList.begin(); iter != theList.end(); iter++)
        *iter = data += 2;
    iter = find(theList.begin(), theList.end(), 8);
    if(iter != theList.end())
        cout << "\n8을 발견하였습니다.\n";
    else
        cout << "\n8을 발견하지 못했습니다.\n";
    return 0;
}
```

find()는 세개인수를 가진다. 처음의 두개는 탐색범위를 지정하는 반복자값들이고 제3인수는 찾으려는 값이다. 여기서는 용기에 값 2, 4, 6, 8, 10을 채워넣는다. 그다음 find()알고리듬에 의하여 수 8을 찾는다. find()가 iList.end()를 돌려주면 일치하는 수를 찾지 못하고 다른 용기의 끝에 도달했다는것을 의미하고 그렇지 않으면 8을 가진 항목이 있어야 한다. 여기서 출력은 다음과 같다.

```
8을 발견하였습니다.
```

그러면 반복자값을 사용하여 8이 용기안의 어디에 보관되였는가를 알수 있는가?

용기의 선두로부터 일치하는 항목의 범위는 (iter - iList.begin())으로부터 계산할수 있다. 그러나 이것은 목록에 사용된 반복자들에 좋은 조작이 아니다. 물론 반복자에는 량방향반복자가 있으므로 그것을 사용하여 산수조작을 할수 있다. vector와 queue에서처럼 직접호출반복자를 사용하여 산수조작을 할수 있다. 따라서 목록 iList나 벡토르 v를 검색한다면 실례 15-19의 마지막 부분을 다음과 같이 고쳐쓸수 있다.

```
iter = find(v.begin(), v.end(), 8);
if(iter != v.end())
  cout << "\n" << (iter - v.begin() << "위치에서 8을 발견하였습니다.";
else
  cout << "\n8을 발견하지 못했습니다.\n";
```

출력은 다음과 같다.

```
3위치에서 8을 발견하였습니다.
```

알고리듬이 반복자를 인수로 사용하는 다른 실례가 있다. 여기서는 copy()알고리듬을 vector에 사용한다. 사용자는 한 벡토르에서 다른 벡토르에로 복사하려는 위치범위를 지정하고 프로그람은 그것들을 복사한다. 반복자는 이 범위를 지정한다.

(실례 15-20) copy()알고리듬에 반복자의 사용

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main()
{
   int beginRange, endRange;
   int arr[] = { 11, 13, 15, 17, 19, 21, 23, 25, 27, 29 };
   vector<int> v1(arr, arr + 10);
   vector<int> v2(10);
   cout << "복사하려는 범위를 입력하시오(례를 들면 2 5):";
   cin >> beginRange >> endRange;
   vector<int>::iterator iter1 = v1.begin() + beginRange;
   vector<int>::iterator iter2 = v1.begin() + endRange;
   vector<int>::iterator iter3;
   iter3 = copy(iter1, iter2, v2.begin());
   iter1 = v2.begin();
   while(iter1 != iter3)
      cout << *iter1++ << ' ';
   cout << endl;
   return 0;
}
```

프로그람의 대화는 다음과 같다.

```
복사하려는 범위를 입력하시오(례를 들면 2 5): 3 6
17  19  21
```

v2의 전체내용이 아니라 복사하려는 항목들의 범위만 표시한다. copy()는 목적용기에 복사된 마지막 항목(실례로 마지막 항목의 다음 항목)을 지정하는 반복자를 돌려준다. 이 경우에는 iter2이다. 프로그람은 이 값을 while순환에 사용하여 복사된 항목들만 표시한다.

제 5 절. 특수반복자

이 절에서는 반복자의 두가지 특수형식을 시험한다. 즉 반복자접속기는 흥미있는 방법으로 반복자의 동작을 변경할수 있고 스트림반복자는 입력과 출력스트림이 반복

자처럼 동작하게 한다.

1. 반복자접속기

STL은 표준반복자의 세가지 변종을 제공한다. 이것은 역반복자, 삽입반복자, 생기억반복자이다. 역반복자(back iterator)는 용기를 역방향으로 순환하게 한다. 삽입반복자(insert iterator)는 copy()와 merge()와 같은 알고리듬의 동작을 변경하여 현존자료를 고쳐쓰는것이 아니라 용기에 자료를 삽입하게 한다. 생기억반복자(raw storage iterator)는 출력반복자가 초기화되지 않은 기억기에 자료를 보관하게 하는 특수한 경우에 사용하고 여기서는 무시한다.

1) 역반복자

용기를 거꾸로 즉 끝에서부터 순환해야 한다. 례를 들면 다음과 같다.

```
list<int>::iterator iter;
iter = iList.end();
while(iter != iList.begin())
    cout << *iter-- << ' ';
```

그러나 이것은 범위가 틀리므로 동작하지 않는다.

반대로 반복하자면 역반복자를 사용해야 한다. 실례 15-21은 역반복자에 의하여 목록의 내용을 반대순서로 표시한다.

(실례 15-21) 역반복자의 사용

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
   int arr[] = { 2, 4, 6, 8, 10 };
   list<int> theList;
   for(int j=0; j<5; j++)
      theList.push_back(arr[j]);
   list<int>::reverse_iterator revIt;
   revIt = theList.rbegin();
   while(revIt != theList.rend())
      cout << *revIt++ << ' ';
   cout << endl;
   return 0;
}
```

역반복자를 사용할 때 성원함수 rbegin()과 rend()를 사용한다.(일반적으로는 정방향반복자와 그것을 사용하는 변수를 사용한다.)

용기의 끝에서 시작하려면 성원함수 rbegin()을 호출해야 하고 역반복자를 증가시켜야 한다. 역반복자를 감소시켜서는 안된다. 즉 revIt--는 동작하지 않는다. reverse_iterator를 사용하여 항상 rbegin()으로부터 rend()에로 증가연산자를 사용하여 이동한다.

2) 삽입반복자

copy()와 같은 일부 알고리듬은 목적용기의 현존내용을 고쳐쓴다. 실례 15-22는 한 deque를 다른 곳에 복사하는 실례이다.

(실례 15-22) deque와 보충복사

```
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
int main()
{
  int arr1[] = { 1, 3, 5, 7, 9 };
  int arr2[] = { 2, 4, 6, 8, 10 };
  deque<int> d1;
  deque<int> d2;
  for(int j=0; j<5; j++)
  {
     d1.push_back(arr1[j]);
     d2.push_back(arr1[j]);
  }
  copy(d1.begin(), d1.end(), d2.begin());
  for(int k=0; k<d2.size(); k++)
     cout << d2[k] << ' ';
  cout << endl;
  return 0;
}
```

그 출력은 다음과 같다.

```
1 3 5 7 9
```

d2의 내용을 d1에 써넣었으므로 d2이 표시되고 그전의 내용은 출력되지 않는다. 그러나 때때로 copy()는 삽입연산자를 사용하여 용기에 새로운 원소들을 삽입한다. 그 방식에는 세가지가 있다.

· 역방향삽입자 back_inserter는 끝에 새로운 항목들을 추가한다.
· 정방향삽입자 front_inserter는 선두에 새로운 항목들을 삽입한다.
· 삽입자 inserter는 주어진 위치에 새로운 항목들을 삽입한다.

실례 15-23은 역방향삽입자를 사용하는 방법을 보여준다.

(실례 15-23) deque에로의 삽입

```
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
int main()
{
  int arr1[] = { 1, 3, 5, 7, 9 };
  int arr2[] = { 2, 4, 6 };
  deque<int> d1;
  deque<int> d2;
  for(int i=0; i<5; i++)
     d1.push_back(arr1[i]);
  for(int j=0; j<3; j++)
     d2.push_back(arr2[j]);
  copy(d1.begin(), d1.end(), back_inserter(d2));
```

```
    cout << "\nd2:";
    for(int k=0; k<d2.size(); k++)
        cout << d2[k] << ' ';
    cout << endl;
    return 0;
}
```

역방향삽입자는 용기의 push_back()성원함수를 사용하여 새로운 항목을 목적용기 d2의 끝에 즉 현존항목들의 뒤에 삽입한다. 이때 원천용기 d1는 변경된다. 프로그람의 출력은 d2의 새로운 내용을 표시한다.

```
2 4 6 1 3 5 7 9
```

정방향삽입자를 지정하면 즉

```
copy(d1.begin(), d1.end(), front_inserter(d2));
```

새로운 항목들이 용기의 앞에 삽입된다. 정방향삽입자의 기초에 놓여있는 기구는 용기의 push_front()성원함수로서 항목들을 용기의 앞에 그 순서를 반전하여 밀어넣는다. 출력은 다음과 같다.

```
9 7 5 3 1 2 4 6
```

또한 삽입반복자의 삽입자판을 사용하여 임의의 원소로부터 시작하여 새로운 항목들을 삽입할수 있다. 실례로 새로운 항목을 d2의 선두에 삽입하려면 다음과 같이 한다.

```
copy(d1.begin(), d1.end(), inserter(d2, d2.begin()));
```

inserter에로의 첫 인수는 복사하려는 용기이고 둘째 인수는 복사를 시작하는 위치를 가리키는 반복자이다. inserter는 용기의 insert()성원함수를 사용하므로 원소들의 순서는 반전되지 않는다. 명령문의 출력결과는 다음과 같다.

```
1 3 5 7 9 2 4 6
```

inserter에로의 제2인수를 변경하여 새로운 자료를 d2의 어디에나 삽입할수 있다.

front_inserter는 push_front()성원함수를 가지지 않는 vector에 사용할수 없고 다만 vector의 끝에서 호출할수 있다.

2. 스트림반복자

스트림반복자는 파일과 입출력장치들을 반복자처럼 취급한다. 스트림반복자는 파일과 입출력장치들을 알고리듬에 대한 인수로서 간단히 사용하게 한다.

입출력반복자클라스의 중요한 목적은 스트림반복자클라스를 지원하는것이다. 입출력반복자는 적당한 알고리듬을 입출력스트림에 직접 사용하게 한다.

사실상 스트림반복자는 각이한 형의 입력 혹은 출력을 위하여 형식화된 클라스들의 객체이다. 두개의 스트림반복자 ostream_iterator와 istream_iterator가 있다.

1) ostream_iterator클라스

ostream_iterator객체는 출력반복자를 지정하는 알고리듬에 인수로 쓰일수 있으며 실례 15-24에서는 copy()의 인수로 사용된다.

(실례 15-24) ostream_iterator

```
#include <iostream>
#include <list>
#include <algorithm>
```

```
using namespace std;
int main()
{
   int arr[] = { 10, 20, 30, 40, 50 };
   list<int> theList;
   for(int j=0; j<5; j++)
      theList.push_back(arr[j]);
   ostream_iterator<int> osIter(cout, ", ");
   cout << "\n목록의 내용:";
   copy(theList.begin(), theList.end(), osIter);
   cout << endl;
   return 0;
}
```

int형값을 읽어들이기 위한 ostream반복자를 정의한다. 구성자에로의 두개 인수는 int값들이 출력될 스트림과 매개 값들을 하나씩 표시할 때 쓰이는 구분문자렬이다. 스트림값은 대체로 파일이름 혹은 cout이다.(여기서는 cout이다.)

cout에 써넣을 때 임의의 문자들로 이루어지는 구분문자렬을 요구한다. 구분문자로서는 반점과 공백을 사용할수 있다.

copy()알고리듬은 cout에 목록의 내용을 복사한다. ostream반복자는 copy()에로의 제3인수로 사용되고 그것이 목적지로 된다.

```
목록의 내용: 10, 20, 30, 40, 50
```

실례 15-25는 ostream반복자를 사용하여 파일에 써넣는 방법을 보여준다.

(실례 15-25) 파일에 대한 ostream_iterator

```
#include <fstream>
#include <list>
#include <algorithm>
using namespace std;
int main()
{
   int arr[] = { 11, 21, 31, 41, 51 };
   list<int> theList;
   for(int j=0; j<5; j++)
      theList.push_back(arr[j]);
   ofstream outFile("ITER.DAT");
   ostream_iterator<int> osIter(outFile, " ");
   copy(theList.begin(), theList.end(), osIter);
   return 0;
}
```

ofstream형의 객체를 정의하고 파일(여기서는 Iter.dat파일)에 련결한다. 이 객체는 ostream_iterator에로의 첫 인수로 쓰인다. 파일에 써넣을 때 문자렬인수에 "--"와 같은 문자가 아니라 공백문자를 사용한다. 이것은 파일로부터 자료를 쉽게 읽어들이게 한다. 여기서는 공백문자를 사용한다.

실례 15-25로부터 표시할만한 출력은 없으나 본문편집기에 의하여 실례 15-25가 창조한 iter.dat파일을 시험할수 있다. 여기에 다음의 자료가 포함되여야 한다.

```
11  21  31  41  51
```

2) istream_iterator클라스

istream_iterator객체는 입력반복자를 지정하는 임의의 알고리듬에 인수로 쓰일수 있다. 실례 15-26은 copy()에로의 처음 두개의 인수로 그 객체를 사용하는 실례를 준다. 프로그람은 사용자에 의해 cin에 입력한 류동소수점수를 읽고 목록에 보관한다.

(실례 15-26) istream_iterator

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
int main()
{
   list<float> fList(5);
   cout << "\n5개의 류동소수점수를 입력하시오(CTRL^Z로 완료):";
   istream_iterator<float> cinIter(cin);
   istream_iterator<float> endOfStream;
   copy(cinIter, endOfStream, fList.begin()); cout << endl;
   ostream_iterator<float> osIter(cout, "--");
   copy(fList.begin(), fList.end(), osIter); cout << endl;
   return 0;
}
```

프로그람과의 대화는 다음과 같다.

```
5개의 류동소수점수를 입력하시오(CTRL^Z로 완료):1.1  2.2  3.3  4.4  5.5
1.1--2.2--3.3--4.4--5.5
```

copy()에 의하여 cin으로부터 오는 자료는 원천이고 목적이 아니므로 복사하려는 자료범위의 시작과 끝을 모두 지정해야 한다. 시작은 cin과 련결된 istream_iterator이고 1인수구성자에 의하여 cinIter로 정의된다.

그러면 범위의 끝을 무엇으로 하겠는가?

istream_iterator의 기정구성자는 여러가지 특수한 역할을 수행한다. 그것은 항상 스트림의 끝을 표시하는 istream_iterator객체를 창조한다. 자료를 입력할 때 사용자는 Ctrl-Z건결합을 입력하는 방법으로 스트림에 파일끝문자를 전송한다. Enter건은 수값들을 구분하지만 파일의 끝이 아니다.

ostream_iterator를 사용하여 목록의 내용을 표시한다. 물론 목록을 표시하는 다른 방법도 있다. istream반복자를 사용하기전뿐아니라 그것을 정의하기전에도 "5개의 류동소수점수를 입력하시오:"와 같은 재촉문을 표시한다. 이 반복자가 정의되면 곧 표시를 그만두고 입력을 기다린다.

실례 15-27은 copy()알고리듬에 입력할 때에도 cin대신에 파일을 사용한다.

(실례 15-27) 파일에 대한 istream_ iterator

```
#include <iostream>
#include <fstream>
#include <list>
#include <algorithm>
using namespace std;
int main()
{
   list<int> iList;
   ifstream inFile("ITER.DAT");
```

```
    istream_iterator<int> fileIter(inFile);
    istream_iterator<int> endOfStream;
    copy(fileIter, endOfStream, back_inserter(iList)); cout << endl;
    ostream_iterator<int> osIter(cout, "--");
    copy(iList.begin(), iList.end(), osIter); cout << endl;
    return 0;
}
```

실례 15-27의 출력은 다음과 같다.

```
2--21--31--41--51--
```

ifstream객체를 Iter.dat파일과 련결하도록 정의하는데 이 파일은 이미 존재하고 자료를 포함하고있다. 실례 15-26에서는 istream반복자에서와 같이 cout를 사용하지 않고 inFile이라고 이름지은 ifstream객체를 사용한다. 스트림의존객체는 같다.

이 프로그람을 좀 변경하여 iList에 자료를 삽입할 때 back_inserter를 사용하게 한다. iList는 몇개의 항목을 입력할지 모르므로 특정한 크기를 가진 용기가 아니라 빈 용기로 정의하고 입력을 읽어들일 때 그것을 사용한다.

제 6 절. 련상용기

선형적인 고정순서로 자료항목들을 보관하는 순차용기(vector, list, deque)를 보았다. 항목의 검색(첨수번호가 알려지지 않거나 용기의 끝에 배치되여있으면)은 용기안의 항목들을 하나씩 순환하는 속도가 느린 처리를 포함한다.

련상용기에서는 항목들을 순서로 배렬하지 않는다. 그대신 주어진 항목을 고속으로 찾을수 있도록 더 복잡한 방법으로 배렬한다. 이 방법은 대체로 나무구조이다. 물론 하쉬표와 같은것도 가능하다. 탐색속도는 련상용기의 기본우점이다.

탐색은 건을 사용하여 진행하는데 건은 보통 수 혹은 문자렬과 같은 단일값이다. 건의 값은 용기안의 객체의 속성 혹은 객체전체일수 있다.

STL의 두가지 기본적인 련상용기는 set와 map이다.

set는 건을 포함하는 객체들을 보관하고 map는 쌍을 보관한다. 여기서 쌍의 첫 부분은 건을 포함하는 객체이고 둘째 부분은 값을 포함하는 객체이다.

set와 map에는 매개 건에 대하여 하나의 실체만 보관할수 있다. 이것은 매개 단어에 한개의 항목을 가지는 사전과 같다. STL에는 이 제한에 따라 set와 map의 두개의 판이 있다. multiset와 multimap는 set, map와 비슷하고 같은 건을 가지는 여러개의 실체를 포함할수 있다.

련상용기는 많은 성원함수들을 다른 용기들과 공유한다. 그러나 일부 알고리듬(즉 lower_bound()와 equal_range())은 련상용기에만 존재한다. 또한 push와 pop계렬(push_back() 등)과 같이 다른 용기에 존재하는 일부 성원함수들이 련상용기에는 없다. 련상용기에서는 원소들을 용기의 시작이나 끝이 아니라 그것들이 정렬된 위치에 삽입하여야 하므로 pop와 push를 사용하지 않는다.

1. set와 multiset

set는 대체로 자료기지의 종업원처럼 사용자정의클라스의 객체들을 보관하는데 쓰인다. 또한 set는 문자렬과 같은 더 간단한 원소들도 보관한다. 그림 15-5에서 이것을

보여준다. 객체들은 차례로 배렬되고 전체 객체는 건이다.

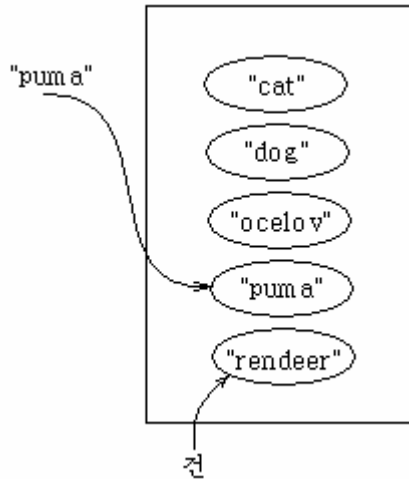



그림 15-5. 문자렬객체들의 모임

첫 실례 15-28은 string클라스의 객체들을 보관하는 set를 보여준다.

(실례 15-28) 문자렬객체를 보관하는 모임

```
#include <iostream>
#include <set>
#pragma warning(disable : 4786) // for set
#include <string>
using namespace std;
int main()
{
   string names[] = { "Kim", "Li", "Pak", "Cha", "Chae" };
   set< string, less<string> > nameSet(names, names+5);
   set< string, less<string> >::iterator iter;
   nameSet.insert("SonU");
   nameSet.insert("Han");
   nameSet.insert("Kang");
   nameSet.insert("Zo");
   nameSet.erase("Cha");
   cout << "\nSize=" << nameSet.size() << endl;
   iter = nameSet.begin();
   while(iter != nameSet.end())
      cout << *iter++ << '\n';
   string searchName;
   cout << "\n검색하려는 이름을 입력하시오:";
   cin >> searchName;
   iter = nameSet.find(searchName);
   if(iter == nameSet.end())
      cout << "모임에는 이름 " << searchName << "이 없습니다.";
   else
      cout << "모임에 이름 " << *iter << "이 있습니다.";
   cout << endl;
```

```
        return 0;
    }
```

set를 정의하자면 보관하려는 객체들의 형(이 경우에 string)과 set의 성원들에 사용되는 함수객체를 지정해야 한다. 여기서는 문자렬객체에 적용할 함수객체로서 less<>()를 사용한다.

set는 다른 STL용기와 류사한 대면부를 가진다. set를 배렬로 초기화할수 있고 insert()성원함수에 의해 모임에 새로운 성원을 삽입할수 있다. set를 표시하려면 그것을 순환해야 한다.

set안에서 특정한 입구를 찾으려면 find()성원함수를 사용해야 한다.(순차용기들에서는 알고리듬판의 find()를 사용한다.) 실례 15-28에서는 탐색하려는 이름으로서 "Kim"을 입력한다.

```
Size=8
Chae
Han
Kang
Kim
Li
Pak
SonU
Zo
검색하려는 이름을 입력하시오:Kim
모임에 이름 Kim이 있습니다.
```

물론 련상용기검색속도의 우점은 이 실례보다 더 많은 실체를 가질 때까지 나타나지 않는다. 여기서 련상용기에만 유효한 한쌍의 중요한 성원함수를 고찰하자. 실례 15-29는 lower_bound()와 upper_bound()의 사용을 보여준다.

(실례 15-29) 모임을 사용하여 범위검사

```
#include <iostream>
#include <set>
#pragma warning(disable : 4786) // for set
#include <string>
using namespace std;
int main()
{
    set< string, less<string> > organic;
    set< string, less<string> >::iterator iter;
    organic.insert("Curina");
    organic.insert("Xanthina");
    organic.insert("Curarina");
    organic.insert("Melamina");
    organic.insert("Cyanimida");
    organic.insert("Phena");
    organic.insert("Aphrodina");
    organic.insert("Imidazole");
    organic.insert("Cinchorine");
    organic.insert("Palmitamide");
    organic.insert("Cyanimida");
```

```
    iter = organic.begin();
    while(iter != organic.end())
       cout << *iter++ << '\n';
    string lower, upper;
    cout << "\n검색하려는 범위(혹은 C Czz)를 입력하시오:";
    cin >> lower >> upper;
    iter = organic.lower_bound(lower);
    while(iter != organic.upper_bound(upper))
       cout << *iter++ << '\n';
    cout << endl;
    return 0;
}
```

우선 프로그람은 유기합성물의 전체모임을 표시한다. 그다음 사용자는 한쌍의 건값을 입력하고 프로그람은 이 범위안의 건들을 표시한다. 여기에 프로그람과의 대화가 있다.

```
Aphrodina
Cinchorine
Curarina
Curina
Cyanimida
Imidazole
Melamina
Palmitamide
Phena
Xanthina
검색하려는 범위(혹은 C Czz)를 입력하시오:Aaa Curb
Aphrodina
Cinchorine
Curarina
```

lower_bound()성원함수는 같은 형의 값인 인수를 건으로 가지고 인수보다 작지 않은 첫 항목에로의 반복자를 돌려준다.(여기서 《작다》의 의미는 set의 정의에서 사용된 함수객체에 의해 결정된다.) upper_bound()함수는 인수보다 큰 첫 입구에로의 반복자를 돌려준다. 물론 이 항목들은 지정된 범위의 값들을 호출하게 한다.

2. map와 multimap

map는 쌍을 보관한다. 쌍은 건객체와 값객체로 이루어진다. 건객체는 검색하려는 건을 포함한다. 값객체는 추가자료를 포함한다. 모임에서처럼 건객체는 문자렬, 수, 복잡한 클라스의 객체일수 있다. 값객체는 보통 문자렬 또는 수일수 있으나 객체 또는 용기일수도 있다.

실례로 건이 단어이고 값은 문서에 단어가 나타나는 회수이며 map는 빈도표일수 있다. 혹은 건은 단어이고 값은 페지수들의 목록이며 이것들의 배렬은 책의 색인을 표시할수 있다. 그림 15-6은 건들이 단어이고 값들은 정의(보통 사전처럼)인 경우를 보여준다.

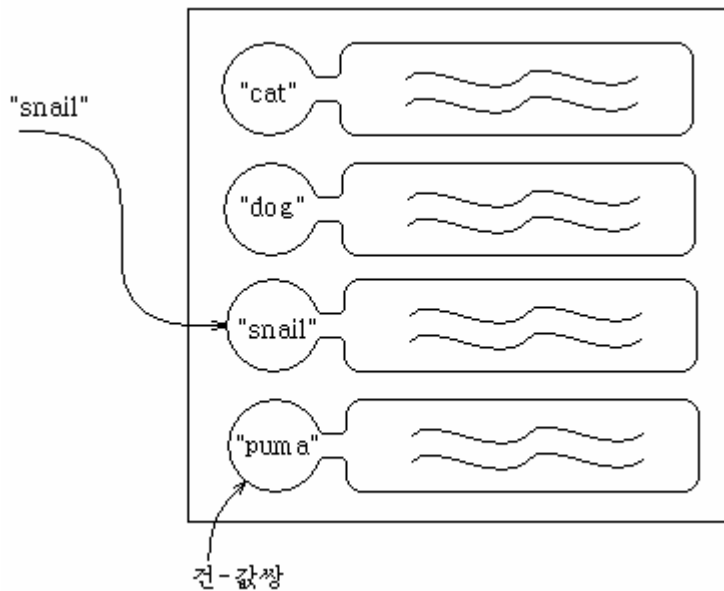



그림 15-6. 단어-구쌍의 map

map를 사용하는 일반적인 방법의 하나는 련상배렬이다. 보통 C++배렬에서 배렬첨수는 특정원소의 호출에 쓰이는 옹근수이다. 식 anArray[3]에서 3은 배렬첨수이다. 련상배렬은 배렬첨수의 자료형을 선택할수 있다는것을 제외하면 배렬과 같은 방법으로 동작한다.

실례로 첨수를 문자렬로 정의하면 anArray["mode"]라고 말할수 있다.

1) 련상배렬

련상배렬로서 사용하는 map의 간단한 실례를 고찰하자. 건은 도시이름이고 값은 도시의 인구수이다. 여기에 실례 15-30이 있다.

(실례 15-30) 련상배렬로서 매프

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main()
{
   string name;
   int pop;
   string states[] = { "PyongYang","HamHung", "ChongJin", "NamPo",
      "WanSan", "KaeSong"};
   int pops[] = { 200, 100, 100, 20, 50, 30 };
   map<string, int, less<string> > mapStates;
   map<string, int, less<string> >::iterator iter;
   for(int j=0; j<6; j++)
   {
      name = states[j];
      pop = pops[j];
```

```
        mapStates[name] = pop;
    }
    cout << "도시이름을 입력하시오:";
    cin >> name;
    pop = mapStates[name];
    cout << "인구: " << pop << "0000\n";
    cout << endl;
    for(iter = mapStates.begin(); iter != mapStates.end(); iter++)
        cout << (*iter).first << ' ' << (*iter).second << ",000\n";
    return 0;
}
```

프로그람을 실행할 때 사용자는 도시이름을 입력한 다음 map를 도시이름을 색인으로 사용하여 검색하고 도시의 인구수를 돌려준다. 끝으로 모든 map안의 이름-인구수쌍을 표시한다. 실행결과는 다음과 같다.

```
도시이름을 입력하시오:ChongJin
인구: 100,000
ChongJin 100,000
HamHung 100,000
KaeSong 30,000
NamPo 20,000
PyongYang 200,000
WanSan 50,000
```

검색속도가 빠른것은 set와 map의 우점이다. 여기서 프로그람은 사용자가 도시이름을 입력하면 그 인구수를 고속으로 검색한다.(수백만개의 자료항목이 있으면 그 의의가 크다.) 도시와 인구수의 목록이 보여주는것처럼 용기를 지나는 순환은 순차용기처럼 고속은 아니지만 아주 효과적이다. 여기서 도시는 자모순으로 정렬된다.

map의 정의는 세개의 형판인수를 가진다.

```
map<string, int, less<string> > mapStates;
```

제1인수는 건의 형으로서 도시이름을 string으로 표시한다. 제2인수는 값의 형으로서 인구수를 1000단위로 표시하는 int이다. 제3인수는 건들에 사용된 순서화방법을 지정하며 여기서는 less<string>()를 사용하여 도시이름에 의하여 자모순으로 정렬한다. 또한 map에로의 반복자를 정의한다.

입력자료는 두개의 개별적인 배렬에 보관된다.(실례에서는 파일로부터 들어온다.)

map에 이 자료를 넣으려면 변수 name과 pop에로 그것을 읽어들이고 명령문

```
mapStates[name] = pop;
```

를 실행하여야 한다.

이것은 일반배렬에로의 삽입처럼 보인다. 그러나 배렬첨수 name은 옹근수가 아니라 문자렬이다.

사용자가 도시이름을 입력하면 프로그람은 명령문

```
pop = mapStates[name];
```

에 의해 적당한 인구수를 찾는다.

배렬첨수처럼 사용하는것외에도 map에서 입구의 두개 부분 즉 건과 값을 반복자를 사용하여 호출할수 있다.

건은 (*iter).first로부터 얻고 값은 (*iter).second로부터 얻는다. 또한 반복자는 다른 용기에서처럼 동작한다.

제 7 절. 사용자정의객체의 보관

지금까지의 실례들에서는 기본형의 객체를 보관한다. 그러나 STL에서 중요한 점은 사용자가 쓴 클라스의 객체를 보관하고 조작할수 있는것이다. 이 절에서는 이것을 수행하는 방법을 설명한다.

1. Person객체모임

개인의 성과 이름, 전화번호를 포함하는 Person클라스를 고찰하자.

Person클라스의 성원들을 창조하고 set에 그것을 삽입하여 전화번호책자료기지를 창조한다. 사용자는 개인의 이름을 입력하여 프로그람과 대화한다. 그다음 set를 검색하고 개인의 자료를 표시한다. 두개이상의 Person객체가 같은 이름을 가질수 있으므로 multiset를 사용한다.

(실례 15-31) Person객체를 multiset에 보관

```
#include <iostream>
#include <set>
#pragma warning (disable : 4786)
#include <string>
using namespace std;
class Person
{
public:
    string lastName;
    string firstName;
    long phoneNumber;
public:
    Person() : lastName("Blank"), firstName("Blank"), phoneNumber(0) {}
    Person(string lana, string fina, long pn) :
         lastName(lana), firstName(fina), phoneNumber(pn) {}
    bool operator<(const Person& p2) const
    {
       if(lastName == p2.lastName)
            return (firstName < p2.firstName) ? true : false;
       return (lastName < p2.lastName) ? true : false;
    }
    bool operator==(const Person& p2) const
    {
       return (lastName == p2.lastName && firstName == p2.firstName)
                                                    ? true : false;
    }
    void Display() const
    {
       cout << endl << lastName << "\t" << firstName << "\t" << phoneNumber;
    }
};
int main()
```

```
{
  Person pers1("Kim", "KangChol", 8435750);
  Person pers2("Cha", "Nam", 5667750);
  Person pers3("Li","KwangSu", 5412750);
  Person pers4("Kang", "WonYong", 8420750);
  Person pers5("Chae", "Kwang", 6322750);
  Person pers6("Zo", "UnGyng", 3255750);
  Person pers7("Kim", "KangChol", 3435750);
  Person pers8("Li", "CholU", 7525750);
  multiset<Person, less<Person> > persSet;
  multiset<Person, less<Person> >::iterator iter;
  persSet.insert(pers1);
  persSet.insert(pers2);
  persSet.insert(pers3);
  persSet.insert(pers4);
  persSet.insert(pers5);
  persSet.insert(pers6);
  persSet.insert(pers7);
  persSet.insert(pers8);
  cout << "\n입구의 개수=" << persSet.size();
  iter = persSet.begin();
  while(iter != persSet.end())
     (*iter++).Display();
  string searchLastName, searchFirstName;
  cout << "\n\n검색하려는 사람의 성을 입력하시오:";
  cin >> searchLastName;
  cout << "\n\n검색하려는 사람의 이름을 입력하시오:";
  cin >> searchFirstName;
  Person searchPerson(searchLastName, searchFirstName, 0);
  int cntPersons = persSet.count(searchPerson);
  cout << "이런 이름을 가진 사람은 " << cntPersons << "명입니다.";
  iter = persSet.lower_bound(searchPerson);
  while(iter != persSet.upper_bound(searchPerson))
     (*iter++).Display();
  cout << endl;
  return 0;
}
```

1) 필요한 성원함수

Person클라스가 STL용기와 작업하려면 몇개의 일반성원함수 즉 기정구성자와 재정의된 <연산자, 재정의된 ==연산자들이 있어야 한다. 이 성원함수들은 set클라스와 여러가지 알고리듬에서 사용된다. 다른 경우에는 다른 성원함수들을 요구한다.(대부분의 클라스들에서 재정의된 대입과 복사구성자, 해체자를 제공한다면 여기서 무시한다.)

재정의된 <와 ==연산자들은 const인수를 사용해야 한다. 일반적으로 그것은 동료함수로 하는것이 좋지만 성원함수들을 사용할수도 있다.

2) 정렬

재정의된 <연산자는 set안의 원소들을 정렬하는 방법을 지정한다. 실례 15-31에서

는 Person의 성에 따라 정렬하고 만일 성이 같으면 이름에 의해 정렬하기 위하여 <연산자를 정의한다.

실례 15-31과의 대화가 있다. 우선 프로그람은 전체 목록을 표시한다. 그런데 multiset에 보관되는 원소들은 자동적으로 정렬된다. 그다음 재촉문에서 사용자가 성 "Kim"다음에 이름 "KwangChol"이라고 입력한다. 목록에는 이런 이름을 가지는 사람이 두명 있으므로 둘다 표시된다.

```
입구의 개수=8
KimKangChol    8435750
Cha Nam     5667750
Li   KwangSu    5412750
Kang     WonYong    8420750
Chae     Kwang   6322750
Zo   UnGyng 3255750
KimKangChol    3435750
Li   CholU   7525750
검색하려는 사람의 성을 입력하시오:Kim
검색하려는 사람의 이름을 입력하시오:KwangChol
이런 이름을 가진 사람은 두명입니다.
```

3) 기본형으로서 클라스의 리용

일단 클라스가 정의된 다음에는 그 객체들은 기본형의 변수들과 같은 방법으로 용기에 의하여 조종된다.

우선 size()성원함수에 의하여 전체 입구수를 표시한다. 그다음 목록을 순환하면서 모든 입구를 표시한다.

multiset에서 lower_bound()와 upper_bound()성원함수들을 사용하여 범위안에 있는 모든 원소들을 표시할수 있다. 실제출력에서 아래한계와 웃한계는 같으므로 같은 이름을 가진 사람은 모두 표시된다. 찾으려는 사람(또는 사람들)과 같은 이름을 가지는 《가설적인》 Person객체를 창조하여야 한다. 그다음 lower_bound()와 upper_bound()함수가 목록안의 요소들과 그 객체를 대조한다.

2. Person객체의 목록

주어진 이름에 의하여 Person을 set나 multiset에서 검색하는데서는 매우 고속이다. 그러나 Person객체를 고속삽입 혹은 고속삭제하는것과 관련된다면 그 대신 목록을 사용해야 한다. 실례 15-32에서 그것을 보여준다.

(실례 15-32) Person객체를 목록에 보관

```
#include <iostream>
#include <list>
#include <algorithm>
#include <string>
using namespace std;
class Person
{
public:
   string lastName;
   string firstName;
```

```
    long phoneNumber;
public:
    Person() : lastName("Blank"), firstName("Blank"), phoneNumber(0) {}
    Person(string lana, string fina, long pn) :
        lastName(lana), firstName(fina), phoneNumber(pn) {}
    bool operator<(const Person& p2) const
    {
        if(lastName == p2.lastName)
            return (firstName < p2.firstName) ? true : false;
        return (lastName < p2.lastName) ? true : false;
    }
    bool operator==(const Person& p2) const
    {
        return (lastName == p2.lastName && firstName == p2.firstName)
                                ? true : false;
    }
    bool operator!=(const Person& p2) const
    { return !(*this  == p2); }
    bool operator>(const Person& p2) const
    { return !(*this < p2) && !(*this == p2); }
    void Display() const
    {
        cout << endl << lastName << "\t" << firstName << "\t" << phoneNumber;
    }
    long GetPhone() const { return phoneNumber; }
};
int main()
{
    list<Person> persList;
    list<Person>::iterator iter1;
    persList.push_back(Person("Kim", "KangChol", 8435750));
    persList.push_back(Person("Cha", "Nam", 5667750));
    persList.push_back(Person("Li", "KwangSu", 5412750));
    persList.push_back(Person("Zo", "UnGyng", 3255750));
    persList.push_back(Person("Kang", "WonYong", 8420750));
    persList.push_back(Person("Chae", "Kwang", 6322750));
    persList.push_back(Person("Kim", "KangChol", 3435750));
    persList.push_back(Person("Ra", "KwangIl", 3255750));
    cout << "\n입구의 개수=" << persList.size();
    iter1 = persList.begin();
    while(iter1 != persList.end())
        (*iter1++).Display();
    string searchLastName, searchFirstName;
    cout << "\n\n검색하려는 사람의 성을 입력하시오:";
    cin >> searchLastName;
    cout << "\n\n검색하려는 사람의 이름을 입력하시오:";
    cin >> searchFirstName;
    Person searchPerson(searchLastName, searchFirstName, 0);
```

```
    iter1 = find(persList.begin(), persList.end(), searchPerson);
    if(iter1 != persList.end())
    {
       cout << "이런 이름을 가진 사람은 ";
       do
       {
          (*iter1).Display();
          ++iter1;
          iter1 = find(iter1, persList.end(), searchPerson);
       } while(iter1 != persList.end());
    }
    else
       cout << "이런 이름을 가진 사람은 없습니다.";
    cout << "\n\n전화번호를 입력하시오(1234567형식):";
    long sNumber;
    cin >> sNumber;
    bool foundOne = false;
    for(iter1 = persList.begin(); iter1 != persList.end(); ++iter1)
    {
       if(sNumber == (*iter1).GetPhone())
       {
          if(!foundOne)
          {
             cout << "이런 전화번호를 가진 사람은 :";
             foundOne = true;
          }
          (*iter1).Display();
       }
    }
    if(!foundOne)
       cout << "이런 전화번호를 가진 사람은 없습니다.";
    cout << endl;
    return 0;
}
```

1) 주어진 이름을 가지는 모든 Person의 검색

set나 map가 아니라 목록을 론의하므로 성원함수 lower_bound()와 upper_bound()를 사용할수 없다. 그대신 주어진 이름을 가지는 Person을 모두 찾아내는데 find()성원함수를 사용한다. find()함수가 일치하는것을 하나 찾으면 처음으로 일치한 Person의 다음부터 시작하여 같은 이름을 가진 Person이 또 있는가를 다시 검색하여야 한다. 이것은 프로그람을 복잡하게 하고 순환에 의하여 find()를 두번 호출하게 한다.

2) 주어진 전화번호를 가지는 모든 사람의 검색

주어진 전화번호를 가진 사람을 찾는것은 이름을 사용하여 찾기보다 더 힘들다. 그것은 find()와 같은 성원함수가 원시적인 속성을 탐색하는데 사용되도록 되여있기때문이다. 이 실례에서는 목록을 순환하면서 찾으려는 전화번호와 목록의 매개 성원들의 전화번호를 수동적으로 비교하면서 전화번호를 검색하는 원시적인 수법을 사용한다.

```
    if(sNumber == (*iter1).GetPhone())
      …
```

우선 프로그람은 모든 항목들을 표시하고 사용자에게 이름을 묻고 일치하는 사람들을 찾는다. 그다음 전화번호를 묻고 다시 일치하는 사람들을 찾는다. 여기에 프로그람과의 대화가 있다.

```
입구의 개수=8
KimKangChol    8435750
Cha Nam    5667750
Li   KwangSu    5412750
Zo   UnGyng 3255750
Kang    WonYong    8420750
Chae    Kwang  6322750
KimKangChol    3435750
Ra   KwangIl 3255750
검색하려는 사람의 성을 입력하시오:Ra
검색하려는 사람의 이름을 입력하시오:KwangIl
이런 이름을 가진 사람은
Ra   KwangIl 3255750
전화번호를 입력하시오(1234567형식): 3255750
이런 전화번호를 가진 사람은 :
Zo   UnGyng 3255750
Ra   KwangIl 3255750
```

실례에서는 주어진 이름을 가지는 사람을 하나 찾고 주어진 전화번호를 가지는 사람을 두명 찾는다. 클라스의 객체들을 보관하는 목록을 사용할 때 그 클라스용의 4개의 비교연산자 ==, !=, <, >를 선언해야 한다. 실제로 사용하는 알고리듬에 따라서 이 연산자들을 모두 정의할 필요는 없다. 이 실례에서는 ==연산자를 정의할 필요가 없으나 모두 정의한다. 목록에서 size()알고리듬을 사용하자면 <연산자를 정의하여야 한다.

제 8 절. 함수객체

함수객체를 STL에서 널리 사용한다. 여기서 중요한것은 알고리듬의 인수로서의 사용이다. 함수객체는 알고리듬의 조작을 전용화하게 한다. 이 장에서 이미 함수객체를 창조하고 실례 15-6에서 사용하였다. 거기서 우리는 자료를 반대순서로 조작하는데 사용하는 미리 정의된 함수객체 greater<>()의 실례를 보았다. 이 절에서는 다른 미리 정의된 함수객체를 시험하고 STL알고리듬이 수행하는 조종을 확장하기 위하여 자체로 함수객체를 쓰는 방법을 고찰한다.

함수객체는 객체처럼 보이도록 클라스안에 포함하는 함수이다. 그러나 클라스에는 자료가 없고 오직 한개의 성원함수 즉 재정의된 연산자가 있다. 클라스는 자주 각이한 형과 작업할수 있게 형판화된다.

1. 미리 정의된 함수객체

미리 정의된 함수객체는 FUNCTIONAL머리부파일에 있다. 그것을 표 15-10에 보여준다. 주요 C++연산자들에 모두 대응하는 함수객체가 있다. 표에서 문자 T는 클라

스로서 사용자가 쓴 클라스 또는 기본형이다. 변수 x와 y는 함수객체에 인수로 넘기는 클라스 T의 객체들을 표시한다.

표 15-10. 미리 정의된 함수객체

함수객체	돌림값
T = plus(T,T)	x + y
T = minus(T,T)	x - y
T = times(T,T)	x * y
T = divide(T,T)	x / y
T = modulus(T,T)	x % y
T = negate(T)	-x
bool = equal_to(T,T)	x == y
bool = not_equal_to(T,T)	x != y
bool = greater(T,T)	x > y
bool = less(T,T)	x < y
bool = greater_equal(T,T)	x >= y
bool = less_equal(T,T)	x <= y
bool = logical_and(T,T)	x && y
bool = logical_or(T,T)	x || y
bool = logical_not(T)	!x

산수연산, 형변환을 위한 함수객체가 있다. 산수함수객체의 실례를 고찰하자. 실례에서는 AirTime이라는 클라스를 사용하여 시, 분(초는 없음)으로 이루어지는 시간값을 표시한다. 이 자료형은 비행장에서 려객기의 착발시간에 적당하다. 실례는 plus<>()함수객체를 사용하는 방법과 용기에서 모든 AirTime값을 사용하는 방법을 보여준다.

(실례 15-33) accumulate(), plus() 함수객체

```
#include <iostream>
#include <list>
#include <numeric>
using namespace std;
class AirTime
{
public:
    int hours;
    int minutes;
public:
    AirTime() : hours(0), minutes(0) {}
    void Display() const { cout << hours << ':' << minutes; }
    AirTime(int h, int m) : hours(h), minutes(m) {}
    void Get()
    {
        char dummy;
        cout << "\n시간을 입력하시오(형식 12:59): ";
        cin >> hours >> dummy >> minutes;
    }
```

```
    AirTime operator+(const AirTime right) const
    {
        int tempH = hours + right.hours;
        int tempM = minutes + right.minutes;
        if(tempM >= 60)
        {
            tempH++;
            tempM -= 60;
        }
        return AirTime(tempH, tempM);
    }
    bool operator==(const AirTime& ar2) const
        { return (hours == ar2.hours) && (minutes == ar2.minutes); }
    bool operator!=(const AirTime& ar2) const
        { return !(*this == ar2); }
    bool operator<(const AirTime& ar2) const
        { return (hours < ar2.hours) || (hours == ar2.hours && minutes
                                                       < ar2.minutes); }
    bool operator>(const AirTime& ar2) const
        { return !(*this == ar2) && !(*this == ar2); }
};
int main()
{
    char answer;
    AirTime temp, sum;
    list<AirTime> airList;
    do
    {
        temp.Get();
        airList.push_back(temp);
        cout << "계속하겠습니까(y/n)?";
        cin >> answer;
    } while(answer != 'n');
    sum = accumulate(airList.begin(), airList.end(), AirTime(0,0),
                                                       plus<AirTime>());
    cout << "\n합=";
    sum.Display();
    cout << endl;
    return 0;
}
```

실례에서는 accumulate()알고리듬을 보여준다. accumulate()함수에는 두개의 판이 있다. 3인수판은 어떤 범위의 값들을 +연산자에 의하여 항상 합한다. 4인수판은 표 15-10에 준 산수함수객체들중 하나를 사용할수 있다.

4인수판 accumulate()의 4인수는 범위의 첫째와 마지막원소의 반복자들과 합의 초기값(대체로 0), 원소들에 적용하려는 조작이다. 실례에서는 plus<>()를 사용하여 그것들을 더하지만 그것들을 덜거나 곱하거나 또는 다른 함수객체를 사용하여 다른 조작을 수행할수도 있다. 여기에 실례 15-33의 대화가 있다.

```
시간을 입력하시오(형식 12:59): 3:45
계속하겠습니까(y/n)?y
시간을 입력하시오(형식 12:59): 5:10
계속하겠습니까(y/n)?y
시간을 입력하시오(형식 12:59): 2:25
계속하겠습니까(y/n)?y
시간을 입력하시오(형식 12:59): 0:55
계속하겠습니까(y/n)?n
합=12:15
```

accumulate()알고리듬은 용기안을 순환하면서 원소들을 더하는것보다 더 쉽고 명백할뿐아니라 효과있다.

plus<>()함수객체는 AirTime클라스용으로 +연산자를 재정의할것을 요구한다. +연산자는 그것이 plus<>()함수객체가 기대하는것이므로 const함수이여야 한다.

다른 산수함수객체도 류사한 방법으로 작업한다. logical_and<>()와 같은 론리함수객체는 이 연산을 수행하는 클라스의 객체에 사용할수 있다.(실례로 bool형변수)

2. 자체의 함수객체의 작성

표준함수객체들중에 자기에게 필요한 조작을 수행하는것이 없다면 자체로 써야 한다. 다음의 실례는 그런 일이 요구되는 두가지 경우를 보여준다. 하나는 sort()알고리듬을 포함하고 다른것은 for_each()알고리듬을 포함한다.

클라스에서 지정된 <연산자에 기초하여 각 묶음의 원소들을 정렬하기는 쉽다. 그러나 객체자체가 아니라 객체에로의 지적자를 포함하는 용기를 정렬하려고 한다면 어떻게 하겠는가?

지적자보관은 큰 객체들인 경우에 객체를 용기에 배치할 때 생기는 복사처리를 피할수 있으므로 효과성을 개선하는 좋은 방법이다. 그러나 지적자를 정렬하면 객체들이 객체의 어떤 속성에 의해서가 아니라 지적자정의에 따라 배렬된다.

지적자들의 용기에서 필요한 작업을 수행하는 sort()알고리듬을 만들려면 자기의 요구에 맞게 자료를 정렬하는 방법을 정의하는 함수객체를 제공하여야 한다.

실례에서는 Person객체에로의 지적자벡토르를 사용한다. Person이 아니라 지적자에 따라서 객체들을 벡토르에 배치하고 보통의 방법으로 정렬한다. 이 경우에는 정렬후에 전혀 변경이 없다. 그것은 항목들이 주소를 가지는 순서로 삽입되기때문이다. 이것은 우리가 요구하는것이 아니다. 다음에 벡토르를 함수객체 ComparePersons()에 의하여 정확히 정렬한다. 이 경우에는 지적자자체가 아니라 지적자의 내용에 따라서 항목들을 정렬한다. 결과는 Person객체들이 이름에 따라 자모순으로 정렬된것이다.

(실례 15-34) Person객체를 지적자에 보관하고 정렬하기

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;
class Person
{
public:
   string lastName;
```

```
    string firstName;
    long phoneNumber;
public:
    Person() : lastName("Blank"), firstName("Blank"), phoneNumber(0) {}
    Person(string lana, string fina, long pn) :
            lastName(lana), firstName(fina), phoneNumber(pn) {}
    bool operator<(const Person& p2) const
    {
        if(lastName == p2.lastName)
            return (firstName < p2.firstName) ? true : false;
        return (lastName < p2.lastName) ? true : false;
    }
    bool operator==(const Person& p2) const
    {
        return (lastName == p2.lastName && firstName == p2.firstName)
                            ? true : false;
    }
    void Display() const
    {
        cout << endl << lastName << "\t" << firstName << "\t" << phoneNumber;
    }
    long GetPhone() const { return phoneNumber; }
};
class ComparePersons
{
public:
    bool operator()(const Person* ptrP1, const Person* ptrP2) const
        { return *ptrP1 < *ptrP2; }
};
class DisplayPersons
{
public:
    void operator()(const Person* ptrP) const
        { ptrP->Display(); }
};
int main()
{
    vector<Person*> vectPtrsPers;
    Person* ptrP1 = new Person("Kim", "KangChol", 8435750);
    Person* ptrP2 = new Person("Cha", "Nam", 5667750);
    Person* ptrP3 = new Person("Li", "KwangSu", 5412750);
    Person* ptrP4 = new Person("Zo", "UnGyng", 3255750);
    Person* ptrP5 = new Person("Kang", "WonYong", 8420750);
    Person* ptrP6 = new Person("Chae", "Kwang", 6322750);
    vectPtrsPers.push_back(ptrP1);
    vectPtrsPers.push_back(ptrP2);
    vectPtrsPers.push_back(ptrP3);
    vectPtrsPers.push_back(ptrP4);
```

```
    vectPtrsPers.push_back(ptrP5);
    vectPtrsPers.push_back(ptrP6);
    for_each(vectPtrsPers.begin(),vectPtrsPers.end(), DisplayPersons());
    sort(vectPtrsPers.begin(), vectPtrsPers.end());
    cout << "\n\n정렬된 지적자들 ";
    for_each(vectPtrsPers.begin(),vectPtrsPers.end(), DisplayPersons());
    sort(vectPtrsPers.begin(), vectPtrsPers.end(), ComparePersons());
    cout << "\n\n정렬된 개인자료 ";
    for_each(vectPtrsPers.begin(),vectPtrsPers.end(), DisplayPersons());
    while(!vectPtrsPers.empty())
    {
       delete vectPtrsPers.back();
       vectPtrsPers.pop_back();
    }
    cout << endl;
    return 0;
 }
```

실례 15-34의 출력은 다음과 같다.

```
Kim KangChol    8435750
Cha Nam     5667750
Li  KwangSu     5412750
Zo  UnGyng 3255750
Kang    WonYong     8420750
Chae    Kwang   6322750
정렬된 지적자들
Kim KangChol    8435750
Chae    Kwang   6322750
Kang    WonYong     8420750
Zo  UnGyng 3255750
Li  KwangSu     5412750
Cha Nam     5667750
정렬된 개인자료
Cha Nam     5667750
Chae    Kwang   6322750
Kang    WonYong     8420750
Kim KangChol    8435750
Li  KwangSu     5412750
Zo  UnGyng 3255750
```

먼저 원래의 순서를 보여주고 지적자에 의해 정렬되지 않은것을 보여주며 끝으로 이름에 의해 정확히 정렬된것을 보여준다.

1) ComparePersons()함수객체

sort()알고리듬의 2인수판은

```
sort(vectPtrsPers.begin(), vectPtrsPers.end());
```

이고 지적자들이 기억기안에서 그 주소에 의해 분류된다. Person객체들을 이름순으로 분류하기 위하여 3인수판의 sort()를, Compare Persons()함수객체를 제3인수로서 사용한다. 즉

```
sort(vectPtrsPers.begin(), vectPtrsPers.end(), ComparePersons());
```

함수객체 ComparePersons()는 실례 15-34에서 다음과 같이 정의된다.

```
// 지적자를 사용하여 Person들을 비교하는 함수객체
class ComparePersons
{
public:
   bool operator()(const Person* ptrP1, const Person* ptrP2) const
      { return *ptrP1 < *ptrP2; }
};
```

operator()는 Person에로의 지적자인 두개의 인수를 가지고 지적자자체가 아니라 그 내용을 비교한다.

2) DisplayPersons()함수객체

이전과는 다른 방법으로 용기내용을 표시한다. 용기를 횡단할 대신에 for_each()함수에 함수객체를 제3인수로 넘긴다.

```
for_each(vectPtrsPers.begin(),vectPtrsPers.end(), DisplayPersons());
```

이것은 DisplayPersons()함수객체가 벡토르안의 매개 객체를 호출하게 한다. 여기에 DisplayPersons()가 있다.

```
// 지적자를 사용하여 Person들을 표시하는 함수객체
class DisplayPersons
{
public:
   void operator()(const Person* ptrP) const
      { ptrP->Display(); }
};
```

인수에 의한 단일함수호출은 벡토르의 모든 Person객체들을 표시한다.

3. 용기의 동작을 변경하는데 쓰이는 함수객체

실례 15-34에서는 알고리듬의 동작을 변경하는데 사용하는 함수객체를 보았다. 또한 함수객체는 용기의 동작을 변경할수 있다. 실례로 객체에로의 지적자들의 모임을 지적자대신에 객체에 기초하여 자동적으로 출력하려고 한다면 용기를 정의할 때 적당한 함수객체를 사용해야 한다.

요 약

이 장에서는 STL에 대한 간단한 개요를 주었다. 그러나 주요항목들을 언급할 때 STL을 사용하기 시작하는데 충분한 정보를 얻었다.

STL은 세개의 기본구성요소 즉 용기, 알고리듬, 반복자로 이루어진다. 용기는 두가지 즉 순차용기와 련상용기로 나누인다. 순차용기는 vector, list, deque이다. 련상용기는 set, map, 그와 밀접히 련관된 multiset와 multimap이다. 알고리듬은 용기에 대

하여 정렬, 복사, 탐색과 같은 조작을 수행한다. 반복자는 용기원소에로의 지적자처럼 동작하고 알고리듬과 용기사이의 접속을 제공한다.

모든 알고리듬이 모든 용기에 적합한것은 아니다. 반복자는 알고리듬과 용기들을 적당히 결합하도록 담보하기 위하여 사용한다. 반복자는 두 종류의 용기용으로 정의되고 알고리듬에 인수로 쓰인다. 용기의 반복자가 알고리듬과 어울리지 않으면 번역프로그람오유가 생긴다.

입출력반복자는 입출력스트림을 직접 련결하고 입출력장치와 용기사이에 자료를 직접 전송한다. 특수반복자는 역방향삽입을 허용하고 일부 알고리듬의 동작을 변경하여 현존자료를 다시 쓰지 않고 자료를 삽입한다.

알고리듬은 여러개의 각이한 용기와 작업하는 독립적인 함수이다. 또한 매개 용기는 자기의 특정한 성원함수를 가진다. 일부 경우에 같은 함수들이 알고리듬에도 있고 성원함수도 있다.

STL용기용의 알고리듬은 <연산자와 같이 일정한 성원함수들이 재정의되여있는 클라스의 객체들과 작업한다. find_if()와 같은 일부 알고리듬의 동작을 함수객체에 의하여 전용화할수 있다. 함수객체는 ()연산자만 포함하는 클라스로부터 실례화된다.

문 제

1. STL용기는

① 클라스 Employee의 객체를 보관하는데

② 고속호출방식으로 원소들을 보관하는데

③ C++프로그람을 번역하는데

④ 기억기에 객체들이 보관되는 방법을 조직하는데 사용할수 있다.

어느것이 옳은가?

2. STL순차용기에는 어떤것들이 있는가?

3. 두개의 주요 STL련상용기를 들어보시오.

4. STL알고리듬은

① 용기에 대하여 조작하는 독자적인 함수이다.

② 성원함수와 용기사이의 련결이다.

③ 적당한 용기클라스들의 동료함수이다.

④ 적당한 용기클라스들의 성원함수이다.

어느것이 옳은가?

5. STL의 반복자의 한가지 목적은 알고리듬과 용기를 련결하는것이다. 옳은가?

6. find()알고리듬은

① 두개의 용기에서 원소들이 일치하는 렬을 찾는다.

② 주어진 용기와 일치하는 용기를 만든다.

③ 처음 두개의 인수들로서 반복자를 가진다.

④ 용기원소들을 처음의 두개 인수로서 가진다.

어느것이 옳은가?

7. 알고리듬은 STL용기에만 사용할수 있는가?

8. 알고리듬에는 보통 어떤 값에 의하여 범위가 주어지는가?

9. 알고리듬의 동작을 전용화할 때 어떤 실체가 자주 쓰이는가?

10. vector는

① 벡토르의 임의의 위치에 많은 새로운 원소들을 삽입하는데

② 항상 용기의 앞에만 새 원소들을 삽입하는데

③ 첨수가 주어지고 대응하는 원소를 고속호출하는데

④ 원소의 건값이 주어지고 대응하는 원소를 고속호출하는데 적합한 용기이다. 어느것이 옳은가?

11. back()성원함수는 용기의 맨뒤에 있는 원소를 삭제한다. 옳은가?

12. 기정구성자를 가진 벡토르 v를 정의하고 크기가 11로 설정되는 1인수구성자를 가지는 벡토르 w를 정의한 다음 push_back()로 이 벡토르들의 매개에 세개의 원소들을 각각 삽입하면 size()성원함수는 v와 w에 대하여 어떤 값을 돌려주는가?

13. unique()알고리듬은 용기로부터 어떤 원소들을 삭제하는가?

14. deque에서

① 자료는 임의의 위치에 고속삽입 혹은 고속삭제할수 있다.

② 자료는 임의의 위치에서 삽입 혹은 삭제할수 있으나 처리는 상대적으로 느리다.

③ 자료는 어느 한 끝에 고속삽입 혹은 삭제할수 있다.

④ 자료는 어느 한 끝에서 삽입 혹은 삭제할수 있으나 처리는 상대적으로 느리다. 어느것이 옳은가?

15. 반복자는 어떤 기능을 수행하는가?

16. 반복자는 항상 용기를 통하여 앞뒤로 이동할수 있는가?

17. 목록에서는 어떤 반복자를 몇개 사용해야 하는가?

18. iter가 용기에서 반복자인 경우에 iter에 의해 지적된 객체의 값을 가지며 그다음 iter가 다음원소를 가리키게 하는 식을 쓰시오.

19. copy()알고리듬은

① 복사원천원소에로의 반복자

② 복사목적원소에로의 반복자

③ 복사원천의 마지막 원소에로의 반복자

④ 복사목적의 마지막 원소에로의 반복자를 돌려준다.

어느것이 옳은가?

20. reverse_iterator를 사용하려면

① end()로 그것을 초기화하는것으로 시작한다.

② rend()로 그것을 초기화하는것으로 시작한다.

③ 용기안에서 뒤로 이동하기 위하여 그것을 증가시킨다.

④ 용기안에서 뒤로 이동하기 위하여 그것을 감소시킨다.

어느것이 옳은가?

21. back_inserter반복자는 항상 새로운 원소가 현존원소들의 앞에 삽입되게 한다. 옳은가?

22. 스트림반복자는 화면과 건반장치, 파일들을 마치 반복자인것처럼 취급하게 한다. 옳은가?

23. ostream_iterator에 제2인수로서 무엇을 지정하는가?

24. 련상용기에서는

① 값들을 분류순으로 보관한다.

② 건들을 분류순으로 보관한다.

③ 분류는 항상 자모순, 혹은 수자순이다.

④ 분류된 내용을 유지하는데 sort()알고리듬을 사용해야 한다.

어느것이 옳은가?

25. 모임을 정의할 때 무엇을 지정하여야 하는가?

26. 모임에서 insert()성원함수는 분류순으로 건을 삽입한다. 옳은가?

27. map는 무엇을 보관한다.

28. map는 같은 건값을 가지는 원소들을 두개이상 가질수 있다. 옳은가?

29. 용기에 객체 대신에 객체에로의 지적자를 보관한다면

① 객체들을 용기안에 보관하기 위해 복사할 필요가 없다.

② 오직 련상용기만 사용할수 있다.

③ 객체특성들을 건으로 사용하여 객체들을 분류할수 없다.

④ 보통 용기는 적은 기억기를 요구한다.

어느것이 옳은가?

30. 자동적인 정렬에 련상용기를 요구한다면 함수객체에 의해 순서화방법을 정의하고 그 함수객체를 용기의 구성자에서 지적할수 있다. 옳은가?

련습문제

1. 사용자가 입력한 류동소수점수배렬에 sort()알고리듬을 적용하고 결과를 표시하는 프로그람을 작성하시오.

2. 사용자가 입력한 단어들의 배렬에 sort()알고리듬을 적용하고 결과를 표시하는

프로그람을 쓰시오. 단어의 삽입에 push_back()를 사용하고 그 표시에 []연산자와 size()를 사용하시오.

3. int값목록을 사용하시오. 두개의 일반반복자(역반복자가 아니다.)를 사용하시오. 즉 하나의 반복자는 목록안에서 정방향으로 이동하고 다른 하나는 역방향으로 이동하며 while순환에서 목록의 내용을 역전하시오. 여러개의 문자를 바꾸는데 swap()알고리듬을 사용하시오.(짝수와 홀수항목에 대해 동작하는가 확인하시오.)

4. Person클라스를 사용하여 Person객체에로의 지적자를 보관하는 multiset를 창조하시오. ComparePerson함수객체를 가지고 multiset를 정의하고 Person을 이름순으로 보관하시오. 6개의 Person을 정의하고 그것을 multiset에 넣고 그 내용을 표시하시오. 여러개의 Person이 같은 이름을 가질수 있다는데 주의하시오.

5. 한개 배렬에는 홀수를 채우고 한개 모임에는 짝수를 채운다. merge()알고리듬을 사용하여 이 용기들을 벡토르에 결합하시오. 벡토르내용을 표시하여 정확히 실행되였는가 보시오.

6. 련습 3에서 두개의 일반반복자들을 용기의 내용을 반전하는데 사용하였다. 이번에는 벡토르에 대하여 한개의 정방향반복자와 두개의 역방향반복자를 사용하여 같은 일감을 수행하시오.

7. 실례 11-33에서 accumulate()알고리듬의 4인수판을 보았다. 3인수판을 사용하여 이 실례를 다시 고찰하시오.

8. copy()알고리듬을 사용하여 용기안에 렬들을 복사할수 있다. 그러나 목적렬이 원천렬에 겹칠 때 주의해야 한다. copy()를 사용하여 배렬안의 다른 위치에 임의의 렬을 복사하는 프로그람을 쓰시오. 사용자가 값 first1, last1, first2을 입력하시오. 그 목적객체를 오른쪽으로가 아니라 왼쪽으로 겹쌓는 렬을 자리옮김할수 있는 프로그람을 쓰시오.(실례로 여러개의 항목들을 1로부터 10으로가 아니라 10으로부터 9로 옮길수 있다.)

9. 표 15-10에서 C++연산자들에 대응하는 함수객체를 보았다. 또한 실례 11-33에서 accumulate()알고리듬과 함수객체 plus<>()를 보았다. 실례들에서는 함수객체에로의 인수들을 볼 필요가 없었다. 그러나 필요할 때 함수객체안에 인수를 줄 필요가 있다. 실례로 문자렬용기(names)안의 특정한 문자렬(searchName)을 검색한다고 가정하자. 그러면

```
ptr = find_if(names.begin(), names.end(), bind2nd(equal_to<string>(),
                                                        searchName);
```

여기서 equal_to<>()와 searchName은 bind2nd()에 대한 인수이다. 이 명령문은 용기에서 searchName과 같은 첫 문자렬에로의 반복자를 돌려준다. 문자렬의 용기에서 문자렬을 검색하시오. 용기안에서 searchName의 위치를 돌려주어야 한다.

10. copy_backward()알고리듬을 사용하여 련습 7에서 서술한 문제를 극복할수 있

다. 즉 임의의 원천이 임의의 목적과 겹쌓이면 왼쪽으로 렬을 옮길수 없다. copy()와 copy_backward()를 사용하여 겹쌓임에 관계없이 용기안의 어디서나 렬을 옮길수 있는 프로그람을 쓰시오.

11. 옹근수들의 원천파일을 목적파일에 스트림반복자를 사용하여 복사하는 프로그람을 작성하시오. while순환을 사용할수 있다. 순환에서 입력반복자로부터 매개 옹근수값을 읽어들여 즉시 출력반복자에 그것을 삽입하면서 두 반복자를 증가시킨다. 실례 15-25에 의해 창조된 Iter.dat파일로서 적당한 원천파일을 만드시오.

12. 빈도수표는 본문파일안의 매개 단어와 단어들이 나타나는 회수를 목록한다. 사용자에 의해 이름이 입력되는 파일을 위한 빈도표를 창조하는 프로그람을 쓰시오. string과 int쌍의 map를 사용하시오. C서고함수 ispunct()를 사용하여(ctype.h) 구두점을 검사함으로써 단어의 끝을 검출할수 있으며 이때 문자렬성원함수 substr()를 사용할수 있다. 또한 tolower()함수는 대문자들을 소문자로 변환한다.

부 록

부록 1. ASCII문자코드

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	SLE
^Q	17	11	◄	CS1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SIB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Char
32	20	sp
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	¦
125	7D	}
126	7E	~
127	7F	⌂†

† ASCII코드 127은 Del코드이다. MS-DOS에서는 ASCII코드의 8(BackSpace)과

같은 효과가 있다. Del코드의 호출에는 Ctrl+BackSpace건을 사용한다.

Dec	Hex	Char
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	₧
159	9F	ƒ

Dec	Hex	Char
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	░
177	B1	▒
178	B2	▓
179	B3	│
180	B4	┤
181	B5	╡
182	B6	╢
183	B7	╖
184	B8	╕
185	B9	╣
186	BA	║
187	BB	╗
188	BC	╝
189	BD	╜
190	BE	╛
191	BF	┐

Dec	Hex	Char
192	C0	└
193	C1	┴
194	C2	┬
195	C3	├
196	C4	─
197	C5	┼
198	C6	╞
199	C7	╟
200	C8	╚
201	C9	╔
202	CA	╩
203	CB	╦
204	CC	╠
205	CD	═
206	CE	╬
207	CF	╧
208	D0	╨
209	D1	╤
210	D2	╥
211	D3	╙
212	D4	╘
213	D5	╒
214	D6	╓
215	D7	╫
216	D8	╪
217	D9	┘
218	DA	┌
219	DB	█
220	DC	▄
221	DD	▌
222	DE	▐
223	DF	▀

Dec	Hex	Char
224	E0	α
225	E1	ß
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
231	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	φ
238	EE	ε
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	∙
250	FA	·
251	FB	√
252	FC	ⁿ
253	FD	²
254	FE	■
255	FF	

"Ctrl"로 이루어지는 건결합은 CTRL건을 누른 상태에서 건을 누른다. ASCII확장문자(128~255)는 ALT건을 누른 상태에서 건반오른쪽에 있는 수값건에서 문자의 10진코드를 눌러서 입력한다.

부록 2. C++예약어

예약어는 C++언어에 고유한 특성을 실현한다.

예약어는 변수 혹은 다른 사용자정의프로그람요소로서 사용할수 없다.

대부분의 예약어는 C와 C++에서 공통이고 C++에 고유한 예약어도 있다.

일부 번역프로그람은 보충적인 예약어를 가지고있다. 확장예약어들은 보통 하나 혹은 두개의 밑줄로 시작된다. 례를 들면 _cdecl 혹은 __int16.

asm auto bool break case catch char class const const_cast continue default delete do double dynamic_cast else enum explicit extern false float for friend goto if inline int long main mutable namespace new operator private protected public register reinterpret_cast return short signed sizeof static static_cast struct switch template this thread throw true try typedef typeid typename union unsigned using uuid virtual void volatile wchar_t while

부록 3. Visual C++

여기서는 Microsoft Visual C++에 의하여 이 책에서 사용하는 응용프로그람의 종류인 콘솔방식응용프로그람을 만드는 방법을 설명한다.

1. 화면요소들

Visual C++는 Visual Studio의 하나의 구성요소이다. Visual Studio는 C++외에도 다른 언어들과도 작업한다.

Visual C++의 창문은 처음에 세개의 부분으로 구성된다. 창문의 왼쪽에는 보기창문이 있다. 보기창문에는 두개의 타브 즉 ClassView와 FileView가 있다. 프로젝트가 있는 경우에 ClassView타브에는 프로그람의 클라스계층구조가 표시되고 FileView에는 프로젝트에서 사용하는 파일들이 표시된다.

계층구조의 +기호를 전개하고 필요한 문서를 마우스로 련속 두번 눌러서 읽어들일수 있다.

화면의 제일 큰 부분은 보통 문서창문이다. 문서창문은 원천파일들의 표시를 비롯한 여러가지 목적에 사용된다. 화면의 아래에는 여러개의 타브 즉 Build, Debug 등이 있는 긴 창문이 놓여있다. 이 창문은 프로그람의 번역과 같은 조작을 수행할 때 통보를 표시한다.

2. 단일파일프로그람

Visual C++를 사용하여 단일파일콘솔프로그람을 건설하여 실행하는 일은 간단하다. 두가지 경우 즉 파일이 이미 존재하는 경우와 파일을 새로 만들어야 하는 경우가 있다.

두 경우에 현재 열린 프로젝트가 없다는것을 확인하는 일부터 시작해야 한다.

File안내를 선택한다. Close Workspace안내항목이 능동상태이면 그것을 눌러서 현재 작업하고있는 작업공간을 닫는다.

1) 현존파일의 건설

이 책의 실례프로그람들과 같은 .CPP원천파일이 이미 존재한다면 File안내의 Open을 선택한다. (Open Workspace을 선택하지 않는다.) Open대화창문에서 적당한 파일로 이행하여 그것을 선택하고 Open단추를 찰칵한다. 그러면 문서창문에 파일이 나타난다.

이 파일을 번역하고 련결하려면 Build안내의 Build를 선택해야 한다. 대화창문이 열리고 Default Project Workspace를 창조하려고 하는가를 묻는다. 여기서 Yes라고 대답한다. 그러면 파일이 번역되고 적당한 서고파일들이 련결된다.

프로그람을 실행하려면 Project안내의 Execute를 선택한다. 제대로 되였으면 콘솔창문에 프로그람의 출력이 나타난다.

프로그람을 완료할 때 Press any key to continue라는 통보가 창문에 표시된다. 번역프로그람은 프로그람의 실행을 완료한 다음에 이 통보문을 출력하게 하여 프로그람의 출력을 사용자가 충분히 오래동안 볼수 있도록 콘솔표시를 유지한다.

MS-DOS로부터 프로그람을 직접 실행할수 있다. Windows 95와 Windows 98에서 Start단추를 누르고 Programs를 선택하고 MS-DOS Prompt항목을 선택하여 MS-DOS창문을 열수 있다. 이 창문에서 C문자와 그뒤에 현재등록부의 이름이 있는 재촉문을 볼수 있다. cd와 새 등록부이름을 입력하여 한 등록부로부터 다른 등록부에로 이동할수 있다. 프로그람을 실행하기 위하여 적당한 .EXE파일이 등록부에 있는가를 확인하고 프로그람의 이름을 확장자없이 입력한다.

2) 새 파일의 창조

자체로 .CPP파일을 작성하려면 File안내의 New를 선택하고 Files타브를 누른다. C++ Source File을 선택하고 OK를 누른다. 그러면 빈 문서창문이 나타난다. 여기에 자기의 프로그람을 입력한다. File안내의 Save As를 선택하고 정확한 등록부로 이행하여 .CPP확장자를 가진 파일이름(례를 들면 MyProg.cpp)를 입력하여 새 파일을 보관한다.

Build안내에서 Build를 선택하고 기정의 작업공간을 사용하겠는가를 묻는 질문에 Yes라고 응답한다. 그러면 프로그람은 번역되고 련결된다.

오유가 있으면 화면아래의 Build창문에 그것이 나타난다. 이 창문은 Build타브를 선택하여 표시할수 있다. 또한 Build창문에서 오유번호(례하면 C2143)우에 유표를 가져가고 F1건을 누르면 오유설명이 문서창문에 표시된다.

오유를 수정하고 "0 error(s), 0 warning(s)"라는 통보가 나타날 때까지 오유수정과 건설과정을 반복한다. 프로그람을 실행하려면 Build안내로부터 Execute를 눌러야 한다.

새 프로그람을 작성하기 전에 File안내에서 Close Workspace를 선택하는것을 잊어서는 안된다. 이미 건설한 프로그람을 열려면 File안내에서 Open Workspace를 선택하고 적당한 등록부로 이행하여 적당한 이름과 .DSW확장자를 가지는 파일을 두번 련속 누르면 된다.

3) 실행시형정보

12장의 일부 실례들은 실행시형정보(Run-Time Type Information)을 사용한다. Visual C++에서는 이 기능이 동작하게 하기 위한 번역프로그람선택을 가능하게 해야 한다. Project안내에서 Settings를 선택하고 C/C++타브를 누른다. Category목록칸에서 C++ Languages를 선택하고 Enable Run-Time Type Information이라는 검사칸을 선택한다.

3. 여러파일프로그람

프로젝트가 여러개의 파일을 가지면 프로그람의 건설은 더 복잡해진다.

1) 프로젝트와 작업공간

Visual C++는 프로젝트보다 한준위 더 높은 추상화로서 작업공간이라는 개념을 사용한다. 작업공간에는 여러개의 프로젝트를 포함할수 있으며 하나의 등록부와 여러개의 환경파일들로 이루어진다. 작업공간안의 매개 프로젝트는 자기의 등록부를 가지며 전체 프로젝트용의 파일들은 작업공간등록부에 놓인다.

이 책의 작은 프로그람들에서는 개념적으로 매개 프로젝트는 자체의 개별적인 작업공간을 가지는것으로 가정하는것이 편리하다.

프로젝트는 개발하고있는 응용프로그람에 대응된다. 프로젝트정보는 응용프로그람을 생성하는데 필요한 파일들과 이 파일들의 결합방법에 대한 정보로 이루어진다. 보통 프로젝트건설결과는 사용자가 실행할수 있는 하나의 .exe파일이다. (물론 .dll파일과 같은 다른 파일도 있다.)

2) 원천파일들이 이미 존재하는 경우

새로운 프로젝트에 포함하려는 파일들이 이미 특정한 등록부에 있다고 가정하자. File안내의 New를 선택하고 New대화창문에서 Projects타브를 누르고 목록에서 Win32 Console Application을 선택한다. 우선 Location칸에서 등록부에로의 경로를 등록부이름없이 입력한다. 다음에 Project Name칸에 파일들을 포함하는 등록부의 이름을 입력한다. (Location칸의 오른쪽에 있는 단추를 찰칵하여 적당한 등록부로 이행할수 있다.) Create New Workspace칸이 선택되였는가 확인하고 OK를 찰칵한다.

실례로 파일들이 C:\Book\Ch13\Elev에 있다면 Locatoin칸에 C:\Book\Ch13라고 입력한 다음 Project Name마당에서 Elev라고 입력한다. 프로젝트이름을 입력하면 그것이 자동적으로 그 위치에 추가된다.

이때 프로젝트와 관련된 여러개의 파일들 즉 확장자 .dsp, .dsw 등을 가지는 파일들이 등록부에 추가된다. 그다음 프로젝트에 원천파일들을 추가한다. 여기에는 .cpp와 .h파일들이 포함된다. Project안내의 Add to Project를 선택하고 파일들을 눌러서 추가하려는 파일들을 선택하고 OK를 찰칵한다. FileView타브를 누르고 +기호를 전개하여 선택한 파일들을 볼수 있다. 또한 ClassView타브를 눌러서 클라스구조와 성원함수들을 볼수 있다.

파일을 열어서 확인하거나 수정하려면 File안내의 Open을 선택하고 그 파일을 선택한다.

3) 프로젝트의 보관, 닫기, 열기

Save Workspace를 선택하여 Close Workspace를 선택하여 파일을 닫는다. File안내의 Open Workspace를 선택하여 적당한 등록부로 가서 .dsw파일을 선택하고 Open을 누르는 방법으로 현존프로젝트를 연다.

4) 번역과 련결

단일파일프로그람에서처럼 여러파일프로그람을 번역, 련결, 실행하는 가장 간단한

방법은 Build안내에서 Execute를 선택하거나 Build안내의 Build를 선택하여 프로그람을 실행하지 않고 번역, 련결하는것이다.

4. 오유수정

3장에서는 오유수정프로그람을 사용하여 순환과정을 고찰할것을 권고하였다. 여기서는 Visual C++로 오유수정하는 방법을 설명한다.

프로그람을 건설한 다음 번역 및 련결프로그람오유를 고착시키고 편집창문안에 프로그람이 표시되였는가 확인한다.

1) 한걸음실행

F10건을 눌러서 오유수정프로그람을 기동한다. 목록의 여백에 황색화살표가 나타나고 main뒤의 열린괄호를 가리킨다.

프로그람의 선두가 아니라 다른 곳을 선택하려면 오유수정을 시작하려는 행에 유표를 가져간다. 그다음 Build안내로부터 Sart Debug를 선택하고 Run to Cursor를 선택한다. 화살표가 순환안의 명령문들을 하나씩 내려가다가 순환의 웃끝으로 넘어가는것을 볼수 있다.

2) 변수감시

화면의 왼쪽아래구석에 Watch창문이 있다. 프로그람을 한걸음씩 실행할 때 변수값들의 변화를 감시하려면 Watch창문에 이 변수들의 이름을 배치한다. 그러기 위하여 원천코드에서 변수이름을 오른단추로 누른다. 안내가 펼쳐지면 이 안내에서 Quick Watch를 선택한다. 이때 Quick Watch대화창문에서 Add Watch를 누른다. 변수와 그 현재값이 Watch창문에 나타난다. 변수가 유효범위를 벗어나면 Watch창문은 변수이름옆의 값대신에 오유를 통보한다.

3) 함수안으로 들어가기

프로그람은 F11건을 리용하여 함수안에 들어갈수 있고 F10건은 함수호출을 실행한다. F11을 사용하여 cout <<와 같은 서고루틴안으로 들어가면 서고루틴의 원천코드를 추적할수 있다.

4) 중지점

중지점은 임의의 위치에서 프로그람의 실행을 중지시킨다. 그것이 왜 필요한가? 이미 Run to Cursor를 선택하여 유표위치까지 프로그람을 실행할수 있다는것을 보았다. 그러나 여러 위치에서 프로그람을 증가시켜야 하는 경우가 있다. 례를 들면 if문의 다음과 그에 대응하는 else문의 다음에서 프로그람을 정지시켜야 할수 있다. 중지점은 필요한만큼 삽입할수 있으므로 이 문제를 간단히 해결한다.

프로그람에 중지점을 삽입하려면 우선 중지점을 배치하려는 행에 유표를 가져간다. 그다음 오른쪽 마우스단추를 누르고 상황안내에서 Insert/Remove Breakpoint를 선택한다. 여백의 왼쪽에 붉은색원이 나타난다. 그러면 프로그람을 실행할 때마다 중지점에서 중지한다. 그다음 코드를 한걸음 실행하여 변수를 시험하거나 다른 중지점까지

실행한다.

중지점을 없애려면 그것을 오른쪽 단추로 누르고 상황안내에서 Remove Breakpoint를 선택한다.

부록 4. Borland C++ Builder

여기서는 C++ Builder로 콘솔방식응용프로그람을 건설하는 방법을 설명한다.

1. C++ Builder에서 실례프로그람의 실행

이 책의 프로그람들을 C++ Builder에서 실행하려면 약간 변경해야 한다.

Windows의 MS-DOS창문에서는 대부분의 실례프로그람들을 변경하지 않고 실현할수 있다. 그러나 C++ Builder에서 Run안내의 Run지령으로 실현하려면 일정한 기간 콘솔창문을 표시하기 위하여 프로그람의 끝에 한개 명령문을 설치해야 한다. 이것은 두걸음으로 실현할수 있다.

· main()의 마지막 return명령문앞에 getch();를 삽입한다. 이것은 프로그람의 출력을 볼수 있게 한다.

· main()의 선두에 #include <conio.h>를 삽입한다. 이것은 getch()에 요구된다. 여러파일프로그람을 만들려고 한다면 main()의 선두에 #include <condefs.h>를 삽입한다.

2. 화면지우기

C++ Builder를 처음 기동할 때 콘솔방식프로그람에 필요없는 화면객체들이 표시된다. 화면의 왼쪽에는 Form1라는 창문이 있는데 닫기(×)단추를 눌러서 그것을 없앤다. 마찬가지로 Object Inspector도 필요없으므로 닫는다.

Untitl.cpp라는 제목의 원천파일이 있는 창문이 있다. 이것은 C++ Builder가 골격프로젝트를 시작한다는것을 의미한다. 그러나 이것은 필요한 프로젝트의 종류가 아니므로 File안내에서 Close All을 누른다.

Component Palette도 필요없다. 이것은 Standard, Additional, Win32와 같은 표시를 가지는 도구띠이다. View안내의 아래에 있는 Component Palette항목을 눌러서 검사상태를 해제하면 다시는 표시되지 않는다.

3. 새로운 프로젝트창조

새로운 프로젝트를 시작하려면 File안내의 New를 선택한다. New Item대화창문에서 New타브를 누른 다음 Console Wizard아이콘을 두번 련속 누른다. 이때 펼쳐지는 대화창문에서 Window Type가 Console이고 Execute Type가 .EXE라는것을 확인한다. Finish를 누르면 Project Source창문에 다음의 원천파일이 표시된다.

```
#include <condefs.h>
#pragra hdrstop
// -------------------------------------------
#pragra argsused
int main(int argc, char **argv)
{
    return 0;
```

```
}
```

이것은 콘솔방식프로그람의 골격이다. 프로그람에서 일부 행은 필요없고 다른 일부 행들을 추가해야 한다. 여기에 골격을 변경한 례가 있다.

```
// test1.cpp
// #include <condefs.h> // 단일파일프로그람에서 필요없다.
// #pragra hdrstop            // 필요없다
#include <iostream>
#include <conio.h>
// ------------------------------------------
// #pragra argsused // 필요없다
//int main(int argc, char **argv)  인수가 필요없다
int main()
{
    cout << “안녕하십니까!” ;
    getch();
    return 0;
}
```

CONDEFS.H파일은 단일파일프로젝트에서 포함할 필요가 없다. 또한 main()에 인수를 포함할 필요도 없다.

본래의 골격프로그람을 실행하면 콘솔창문이 충분히 오래동안 표시되지 않는다. 그러므로 명령문 getch();를 프로그람의 마지막 return명령문앞에 삽입하여 창문을 정지시킬수 있다. 이것은 프로그람이 건을 누를 때까지 기다리게 하므로 콘솔창문은 사용자가 건을 누를 때까지 보기에 남아있고 getch()함수는 CONIO.H머리부파일을 요구하므로 프로그람의 선두에 포함하여야 한다.

자체의 프로그람을 작성한다면 골격프로그람으로 시작하여 자체의 코드를 입력해야 한다.

4. 프로젝트의 보존

Project Source창문에 표시되는 본문은 확장자 .CPP를 가지는 원천파일이다. 프로젝트에 대한 정보는 확장자 .BPR를 가지는 파일에 기록된다. 따라서 프로젝트를 보관할 때 .CPP파일과 .BPR파일을 둘다 보관한다. 프로젝트를 처음으로 창조할 때 Project1이라고 이름지어진다.

프로젝트의 이름을 변경하여 보관하려면 File안내의 Save Project를 선택하고 보관하려는 위치까지 이동하여 프로젝트의 이름과 .BPR확장자를 입력하고 OK를 누른다.

5. 현존파일로 시작

이미 존재하는 파일들로 프로젝트를 시작할수 있다. 프로젝트에 main()을 포함하는 기본파일이 프로젝트와 같은 이름을 가지게 하려고 한다. C++ Builder는 이러한 이름을 가진 골격파일을 자동생성한다. 프로젝트를 보관하려고 할 때 작성자의 기본파일은 골격파일과 교체된다.

이 문제를 해결하는 방법이 있다. 프로젝트를 MyProj, 기본파일이 MyProj.cpp라고 할 때 다음과 같이 실현한다.

· 기본파일(MyProj.cpp)의 이름을 그 프로젝트이름과 다른 이름(말하자면 XMyProj.cpp)으로 변경한다.

· Save Project로 프로젝트를 보관한다. 프로젝트에 원천파일과 같은 이름과 확장자 .BPR 즉 MyProj.BPR를 주고 Save를 누른다. 골격파일 MyProj.cpp가 생성되고 보관된다. Close All하여 프로젝트를 닫는다.

골격파일 MyProj.cpp를 삭제한다. 원찬파일(XMyProj.cpp)의 이름을 프로젝트와 같은 이름(MyProj.cpp)으로 바꾼다.

그리고 File안내의 Open Project로 프로젝트를 다시 열면 자기 원천파일이 프로젝트의 원천파일로 된다.

6. 번역, 련결, 실행

실행가능프로그람을 건설하려면 Project안내의 Make나 Build를 누른다. 그러면 .CPP파일이 .OBJ파일로 번역되고, .OBJ파일들이 .EXE파일에 결합된다.

1) C++ Builder로부터 실행

앞에서 서술한것처럼 getch()를 삽입하여 프로그람을 수정하였으면 C++ Builder에서 Run안내의 Run을 간단히 선택하여 프로그람을 번역, 련결, 실행할수 있다. 오유가 없으면 콘솔창문이 나타나고 프로그람의 출력이 표시된다.

Visual C++에서처럼 MS-DOS로부터 실행할수도 있다.

2) 사전번역된 머리부파일

Project안내의 Options를 선택하고 Compiler타브를 선택하고 Use Precompiled Headers를 누름으로써 번역시간을 단축할수 있다. 짧은 프로그람에서 번역시간의 대부분은 iostream과 같은 C++머리부파일을 번역하는데 소비된다. Precompiled Headers선택을 사용하면 머리부파일을 처음에 한번만 번역하게 한다.

3) 프로젝트의 닫기와 열기

프로젝트와의 작업이 끝나면 File안내의 Close All을 선택하여 그것을 닫는다.

이미 보관된 프로젝트를 열려면 File안내에서 Open Project를 선택하고 정확한 .BPR파일로 이동하여 그것을 마우스로 련속 두번 누른다.

7. 여러원천파일을 가지는 프로젝트

실제의 응용프로그람은 여러개의 원천(.CPP)파일들로 이루어진다.

C++ Builder에서는 원천파일을 보통 단위(unit)라고 부른다. 대부분의 C++개발환경에서 파일들을 모듈이라고도 부른다. 프로젝트에 1개이상의 원천파일을 사용하려면 main()을 포함하는 기본원천파일에 파일 condefs.h를 포함해야 한다. 즉

```
#include <condefs.h>    // 여러파일프로그람에서 요구된다.
```

1) 추가적인 원천파일의 참조

.CPP파일을 추가할수 있다. File/New를 선택하고 New대화창문에서 Text아이콘을 두번 누른다. 원천코드를 입력하고 Save As를 사용하여 보관한다.

Save As를 리용할 때 Save File As Type목록에서 C++ Builder Unit(.CPP)를 선택한다. 이것은 파일이름에 자동적으로 .cpp확장자가 붙게 한다. 여기서 실패하고 단순히 이름뒤에 .cpp라고 입력하면 그 파일은 C++ Builder의 단위로 인식되지 않는다.

2) 프로젝트에 원천파일의 추가

프로젝트에 원천파일을 추가하려면 Project안내의 Add To Project를 선택하고 적당한 등록부로 가서 파일이름을 선택하고 Open을 누른다.

편집창문에 여러원천파일이 표시되므로 파일들을 고속으로 절환할수 있다.

3) Project Manager

View안내의 Project Manager를 선택하여 어느 원천파일이 프로젝트에 포함되여 있는가를 확인할수 있다. 또한 Windows Explorer에서와 같은 파일관련도를 볼수 있다. 프로젝트아이콘옆에 있는 +기호를 누르면 프로젝트의 모든 원천파일들이 표시된다.

Project Manager안의 파일을 오른쪽 단추로 누르면 Open, Save, Save As, Compile선택이 포함된 상황안내가 표시된다. 이것을 리용하면 개별적인 원천파일들을 쉽게 조작할수 있다.

여러파일프로그람에서는 Project안내에서 Compile Unit를 선택하여 개별적으로 번역할수 있다. Project안내의 Make를 선택하여 모든 원천파일들을 번역, 련결할수 있다.

여러파일프로그람을 번역할 때 C++ Builder는 자동적으로 원시원천파일의 원천코드에 행들을 삽입한다. 이 행들은 다른 원천파일들을 지정한다. 실례로 File1.cpp와 File2.cpp를 포함하는 2진파일프로그람이 있다면 File1.cpp에서 다음행을 볼수 있다.

```
//-----------------------------------------
USEUNIT( “file2.cpp” )
```

이것은 원천파일에 대한 영구적인 변경으로서 여러파일프로그람을 번역하는 좋은 수법이 아니지만 자체로 이 행들을 추가해서는 안된다.

8. 오유수정

C++ Builder에서 오유수정방법을 설명한다.

1) 한걸음실행

F8건을 눌러서 오유수정프로그람을 기동한다. 프로그람이 다시 번역되고 프로그람의 첫행 보통 main()선언자가 강조표시된다. F8건을 반복하여 누르면 프로그람의 매개 명령문에로 조종이 차례로 넘어간다.

2) 변수의 감시

프로그람을 한걸음 실행할 때 변수값들의 변화를 확인하려면 Run안내의 Add Watch를 선택한다. Watch Properties대화창문이 나타나면 Expression칸에 감시하려

는 변수이름을 입력하고 OK를 누른다. Watch List라는 창문이 펼쳐진다. Add Watch 대화창문을 반복 사용하여 Watch List에 감시하려는 변수들을 모두 추가한다. Edit Window와 Watch List가 동시에 있으면 프로그람을 한걸음 실행할 때 변수값들의 변화를 감시할수 있다.

순환안에서 정의된 변수는 감시기구가 인식하지 못하므로 변수를 순환밖에서 정의하도록 프로그람을 고쳐야 한다.

3) 함수안에서 추적

프로그람에서 함수를 사용한다면 F7건을 눌러서 함수안으로 추적해들어갈수 있다. F8건은 함수호출에서 벗어나게 한다.

4) 중지점

프로그람에 중지점을 삽입하는것은 간단하다. 편집창문에 프로그람을 표시한다. 그러면 실행가능한 프로그람행의 반대쪽 왼쪽 여백에 점이 나타난다. 중지점을 삽입하려고 하는 점을 왼쪽 단추로 누르면 왼쪽여백에 붉은색 원이 표시되고 프로그람행이 강조표시된다. 그러면 프로그람을 실행할 때마다 중지점에서 중지한다.

중지점을 삭제하려면 중지점을 다시 왼쪽단추로 누른다.

부록 5. KDevelop

여기서는 우리식 조작체계 《붉은별》과 Linux조작체계에서 통합개발환경 KDevelop를 통하여 콘솔프로그람들을 작성하고 번역, 련결, 실행하는 방법을 설명한다.

1. 프로젝트의 작성과 건설 및 실행

KDevelop에서 Project안내의 New Project를 눌러 응용프로그람위자드를 기동한다.

사용하려는 프로그람언어로서 C++를 선택하고 응용프로그람의 형으로서 Simple Hello world program을 선택한다.

속성안내에서 응용프로그람이름, 응용프로그람을 보관하는 등록부와 같은 일반정보를 입력한다.

CVS와 같은 판조종체계를 선택하고 필요한 자료를 입력한다.

초기의 머리부와 원천파일들을 위한 형판을 설정한다.

그러면 응용프로그람위자드에 의하여 다음과 같은 골격파일이 작성된다.

```
#ifdef HAVE_CONFIG_H
#include <conFIG.h>
#endif

#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char **argv)
{
    cout << “Hello, world!” << endl;
    return EXIT_SUCCESS;
}
```

골격프로그람에서 필요없는 코드를 삭제하고 이 책의 실례프로그람들을 입력한다.

다음에 Build안내의 Build Project를 선택하거나 F8건을 눌러서 프로그람을 건설한다.

오유가 없으면 Build안내의 Execute Program을 선택하여 프로그람을 실행한다.

2. 오유수정

C/C++에 대하여 KDevelop는 편집기에 직접 통합된 내부오유수정프로그람을 제공한다.

오유수정프로그람은 Kdevelop에서의 Debug안내의 Start를 선택하거나 Ctrl+Shift+F9건을 눌러서 기동할수 있다.

Variables창문은 프로그람의 현재 실행점에 있는 모든 국부변수들의 값을 표시한다.

또한 감시변수들도 포함되여있다. 국부변수와 대역변수들을 모두 감시할수 있다. 감시항목을 선택하고 Add단추를 누르거나 Enter건을 눌러서 추가할수 있으며 상황안내에서 그것을 삭제할수도 있다.

부록 6. C/C++의 연산자

다음의 표에 C/C++의 연산자를 종류별로 보여준다.

종류	연산자
범위 해결	범위해결 ::
뒤붙이	배렬요소 [], 함수호출 (), 형강제형변환 (), 성원선택 . 혹은 ->, 뒤붙이식 증가 ++, 뒤붙이식 감소 --
단항	간접 *, 주소 &, 론리적 !, 1의 보수 ~, 앞붙이식 증가 ++, 앞붙이식 감소 --, sizeof, new, delete
반점	반점 ,
곱하기	곱하기 *, 나누기 /, 나머지 %
더하기	더하기 +, 덜기 -
자리밀기	왼쪽 자리밀기 <<, 오른쪽 자리밀기 >>
비교	작기 <, 같거나 작기 <=, 크기 >, 같거나 크기 >=, 같기 ==, 같지않기 !=
비트처리	비트론리적 &, 비트배타론리합 ^, 비트론리합 |
론리	론리적 &&, 론리합 ||
대입	대입 =, 더하기대입 +=, 덜기대입 -=, 곱하기대입 *=, 나누기대입 /=, 나머지대입 %=, 왼쪽 자리밀기대입 <<=, 오른쪽 자리밀기대입 >>=, 비트론리적대입 &=, 비트배타론리합대입 ^=, 비트론리합 |=
조건	조건 ? :
성원에로의 지적자	성원에로의 지적자 * 혹은 ->*
참고	참고 &

부록 7. 연산자의 우선순위와 결합방법

기호[1]	연산자의 형	결합방법
[ ] ( ) . -> 뒤붙이형++와 뒤붙이형--	식	왼쪽에서 오른쪽으로
앞붙이형++와 앞붙이형-- sizeof & * + - ~ !	단항	오른쪽에서 왼쪽으로
형강제형변환연산자들	단항	오른쪽에서 왼쪽으로
* / %	곱하기	왼쪽에서 오른쪽으로
+ -	더하기	왼쪽에서 오른쪽으로
<< >>	비트밀기	왼쪽에서 오른쪽으로
< > <= >=	관계	왼쪽에서 오른쪽으로
== !=	동등	왼쪽에서 오른쪽으로
&	비트론리적	왼쪽에서 오른쪽으로
^	비트배타론리합	왼쪽에서 오른쪽으로
|	비트론리합	왼쪽에서 오른쪽으로
&&	론리적	왼쪽에서 오른쪽으로
||	론리합	왼쪽에서 오른쪽으로
? :	조건식	오른쪽에서 왼쪽으로
= *= /= %= += -= <<= >>= &= ^= |=	단순대입과 복합대입[2]	오른쪽에서 왼쪽으로
,	순차평가	왼쪽에서 오른쪽으로

[1] 연산자들은 우선순위가 감소하는 순서로 배렬하였다. 같은 행 또는 같은 조에 표시된 연산자들의 우선순위는 같다.

[2] 단순대입 혹은 복합대입명령문들의 우선순위는 모두 같다.

부록 8. 표준형판서고 STL

1. 알고리듬

다음의 표에는 STL의 알고리듬을 보여준다. 첫째 란에 함수이름, 둘째 란에 알고리듬의 목적, 셋째 란에 인수를 설명한다.

인수에서 first, last, first1, last1, first2, last2, first3, last3, middle은 용기안의 위치에로의 반복자를 표시한다. 수자가 있는 이름(first1 등)은 여러개의 용기를 구별하는데 쓰인다.

first1, last1은 범위 1을, first2, last2은 범위 2를 구별한다.

인수 function, predicate, op, comp는 함수객체이다. 인수 value, old, new, a, b, init는 용기에 보관된 객체의 값이다. 이 값들은 < 혹은 ==연산자 혹은 comp함수객체에 의하여 정렬되거나 비교된다. 인수 n은 옹근수이다.

목적칸에서 이동하는 반복자는 iter, iter1, iter2에 의하여 지적된다. ier1과 iter2이 모두 쓰일 때에는 매개 용기들(또는 같은 용기안에서 두개의 다른 범위)을 통하여 모두 한걸음씩 이동하는것으로 가정한다.

이 름	목 적	인 수
가변렬조작		
for_each	매개 객체에 function을 적용한다.	first, last, function
find	value와 같은 첫 객체에로의 반복자를 돌려준다.	first, last, value
find_if	predicate가 참인 첫 객체에로의 반복자를 돌려준다.	first, last, predicate
adjacent_find	같은 첫 린접쌍에로의 반복자를 돌려준다.	first, last
adjacent_find	predicate를 만족시키는 첫 린접쌍에로의 반복자를 돌려준다.	first, last, predicate
count	value와 같은 객체의 개수를 n에 대입한다.	first, last, value, n
count_if	predicate를 만족시키는 객체의 개수를 n에 대입한다.	first, last, predicate, n
mismatch	두 범위에서 대응하는 객체들의 같지 않은 첫 쌍을 돌려준다.	first1, last1, first2
mismatch	두 범위에서 predicate를 만족시키지 않는 대응하는 객체들의 첫 쌍을 돌려준다.	first1, last1, first2, predicate
equal	두 범위의 대응하는 객체들이 모두 같으면 참을 돌려준다.	first1, last1, first2
equal	두 범위의 대응하는 객체들이 predicate를 만족시키면 참을 돌려준다.	first1, last1, first2, predicate
search	둘째 범위가 첫째 범위에 포함되면 그 선두를 돌려주고 그렇지 않으면 last1을 돌려준다.	first1, last1, first2, last2

이 름	목 적	인 수
search	둘째 범위가 첫째 범위에서 predicate를 만족시키면 그 선두를 돌려주고 그렇지 않으면 last1을 돌려준다.	first1, last1, first2, last2, predicate
copy	범위1로부터 범위2에로 객체들을 복사한다.	first1, last1, first2
copy_backward	범위1로부터 범위2에로 객체들을 복사한다. 이때 last2로부터 first2에로 반대로 삽입한다.	first1, last1, first2
swap	두 객체를 교체한다.	a, b
iter_swap	두 반복자로 지적한 객체들을 교체한다.	iter1, iter2
swap_ranges	두 범위의 대응하는 객체들을 교체한다.	first1, last1, first2
transform	operator를 적용하여 범위1의 객체들을 범위2의 새로운 객체들로 변환한다.	first1, last1, first2, operator
transform	operator를 적용하여 범위1과 범위2의 객체들을 범위3의 새로운 객체들로 결합한다.	first1, last1, first2, first3, operator
replace	old와 같은 모든 객체들을 new와 같은 객체들로 교체한다.	first, last, old, new
replace_if	predicate를 만족시키는 모든 객체들을 new와 같은 객체들로 교체한다.	first, last, predicate, new
replace_copy	old와 같은 모든 객체들을 new와 같은 객체들로 교체하면서 범위1로부터 범위2로 복사한다.	first, last, first2, old, new
replace_copy_if	predicate를 만족시키는 모든 객체들을 new와 같은 객체들로 교체하면서 범위1로부터 범위2로 복사한다.	first, last, first2, predicate, new
fill	범위의 모든 객체들에 value를 대입한다.	first, last, value
fill_n	first로부터 first+n의 모든 객체들에 value를 대입한다.	first, n, value
generate	함수 gen의 호출에 의해 생성된 값들로 범위를 채운다.	first, last, gen
generate_n	함수 gen의 호출에 의해 생성된 값들로 first로부터 first+n을 채운다.	first, n, gen
remove	범위에서 value와 같은 객체들을 삭제한다.	first, last, value
remove_if	범위에서 predicate를 만족시키는 객체들을 삭제한다.	first, last, predicate
remove_copy	범위1에서 범위2에로 value와 같지 않은 객체들을 복사한다.	first1, last1, first2, value
remove_copy_if	범위1에서 범위2에로 pred를 만족시키는것을 제외한 객체들을 복사한다.	first1, last1, first2, pred
unique	같은 객체들이 련이어 놓여있는 경우에 첫 객체를 제외한 모든것을 삭제한다.	first, last
unique	predicate를 만족시키는 객체들이 련이어 놓여있는 경우에 첫 객체를 제외한 모든것을 삭제한다.	first, last, predicate
unique_copy	범위1로부터 범위2로 객체들을 복사한다. 이때 같은 객체들이 련이어 놓여있는 경우에 첫 객체를 내놓고 모두 삭제한다.	first1, last1, first2
unique_copy	범위1로부터 범위2로 객체들을 복사한다. 이때	first1, last1, first2,

이 름	목 적	인 수
	predicate를 만족시키는 객체들이 련이어 놓여있는 경우에 첫 객체를 제외하고 모두 삭제한다.	predicate
reverse	범위에서 객체들의 렬을 반전한다.	first, last
reverse_copy	범위에서 객체들의 렬을 반전하면서 범위1을 범위2에로 복사한다.	first1, last1, first2
rotate	반복자, middle주위의 렬을 회전시킨다.	first, last, middle
rotate_copy	반복자, middle주위의 렬을 회전시키면서 범위1에서 범위2에로 객체들을 복사한다.	first1, middle, last1, first2
random_shuffle	범위안의 객체들을 우연히 뒤섞는다.	first, last
random_shuffle	우연수함수 rand를 사용하여 범위안의 객체들을 뒤섞는다.	first, last, rand
partition	predicate를 만족시키는 모든 객체들을 그것을 만족시키지 않는것들의 앞으로 이동한다.	first, last, predicate
stable_partition	predicate를 만족시키는 모든 객체들을 그것을 만족시키지 않는것들의 앞으로 이동한다. 또한 두 그룹에서 상대적인 순서를 보존한다.	first, last, predicate
정렬 및 관련조작		
sort	범위의 객체들을 정렬한다.	first, last
sort	비교함수 comp를 사용하여 범위의 원소들을 정렬한다.	first, last, comp
stable_sort	같은 원소들의 순서를 유지하면서 범위의 객체들을 정렬한다.	first, last
stable_sort	같은 원소들의 순서를 유지하면서 범위의 객체들을 비교함수 comp를 사용하여 정렬한다.	first, last, comp
partial_sort	범위의 모든 객체들을 정렬한다. 이때 정렬된 값들은 first와 middle사이에 놓인다. 그리고 middle과 last사이의 객체들의 순서는 정의되지 않는다.	first, middle, last
partial_sort	범위의 모든 객체들을 정렬한다. 이때 정렬된 값들은 first와 middle사이에 놓인다. 그리고 middle과 last사이의 객체들의 순서는 정의되지 않는다. predicate로 순서를 정의한다.	first, middle, last, predicate
partial_sort_copy	partial_sort(first, middle, last)와 같은데 범위2에 결과렬이 배치된다.	first1, last1, first2, last2
partial_sort_copy	partial_sort(first, middle, last, predicate)와 같은데 범위2에 결과렬이 배치된다.	first1, last1, first2, last2, comp
nth_element	nth객체의 전체범위를 정렬한 상태에서 그것이 차지하는 위치에 배치한다.	first, nth, last
nth_element	nth객체의 전체범위를 비교함수 comp를 사용하여 정렬한 상태에서 그것이 차지하는 위치에 배치한다.	first, nth, last, comp
lower_bound	value가 순서를 위반하지 않고 삽입되는 첫 위치에로의 반복자를 돌려준다.	first, last, value
lower_bound	value가 comp에 기초한 순서를 위반하지 않고 삽입되는 첫	first, last, value,

이　름	목　적	인　수
	위치에로의 반복자를 돌려준다.	comp
upper_bound	value가 순서를 위반하지 않고 삽입되는 마지막 위치에로의 반복자를 돌려준다.	first, last, value
upper_bound	value가 comp에 기초한 순서를 위반하지 않고 삽입되는 마지막 위치에로의 반복자를 돌려준다.	first, last, value, comp
eqaul_range	value가 순서를 위반하지 않고 삽입되는 상한과 하한을 포함하는 쌍을 돌려준다.	first, last, value
eqaul_range	value가 comp에 기초한 순서를 위반하지 않고 삽입되는 상한과 하한을 포함하는 쌍을 돌려준다.	first, last, value, comp
binary_search	value가 범위에 있으면 참을 돌려준다.	first, last, value
binary_search	value가 comp에 의해 순서가 결정되는 범위에 있으면 참을 돌려준다.	first, last, value, comp
merge	정렬된 범위1과 범위2를 정렬된 범위3으로 결합한다.	first1, last1, first2, last2, first3
merge	정렬된 범위1과 범위2를 정렬된 범위3으로 결합한다. 이때 순서는 comp에 의해 결정된다.	first1, last1, first2, last2, first3, comp
inplace_merge	두개의 련속 놓이는 정렬된 범위 first, middle과 middle, last를 first, last로 결합한다.	first, middle, last
inplace_merge	두개의 련속 놓이는 정렬된 범위 first, middle과 middle, last를 first, last로 결합한다. 이때 순서는 comp에 의해 결정된다.	first, middle, last, comp
includes	범위 first2, last2의 매개 객체가 범위 first1, last1에 있으면 참을 돌려준다.	first1, last1, first2, last2
includes	범위 first2, last2의 매개 객체가 범위 first1, last1에 있으면 참을 돌려준다. 이때 순서는 comp에 의해 결정된다.	first1, last1, first2, last2, comp
set_union	범위1과 범위2의 원소들의 정렬된 합을 구성한다.	first1, last1, first2, last2, first3
set_union	범위1과 범위2의 원소들의 정렬된 합을 구성한다. 이때 순서는 comp에 의해 결정된다.	first1, last1, first2, last2, first3, comp
set_intersection	범위1과 범위2의 원소들의 정렬된 적을 구성한다. (set와 multiset만)	first1, last1, first2, last2, first3
set_intersection	범위1과 범위2의 원소들의 정렬된 적을 구성한다. 이때 순서는 comp에 의해 결정된다. (set와 multiset만)	first1, last1, first2, last2, first3, comp
set_difference	범위1과 범위2의 원소들의 정렬된 차를 구성한다. (set와 multiset만)	first1, last1, first2, last2, first3
set_difference	범위1과 범위2의 원소들의 정렬된 차를 구성한다. 이때 순서는 comp에 의해 결정된다. (set와 multiset만)	first1, last1, first2, last2, first3, comp

이 름	목 적	인 수
set_symmetric_difference	범위1과 범위2의 원소들의 정렬된 대칭차를 구성한다.(set와 multiset만)	first1, last1, first2, last2, first3
set_symmetric_difference	범위1과 범위2의 원소들의 정렬된 대칭차를 구성한다. 이때 순서는 comp에 의해 결정된다. (set와 multiset만)	first1, last1, first2, last2, first3, comp
push_heap	값을 last-1로부터 범위 first, last의 히프에 배치한다.	first, last
push_heap	값을 last-1로부터 범위 first, last의 히프에 배치한다. 이때 순서는 comp에 의해 결정된다.	first, last, comp
pop_heap	first와 last-1의 값들을 교체한다. first와 last-1을 히프로 만든다.	first, last
pop_heap	first와 last-1의 값들을 교체한다. first와 last-1을 히프로 만든다. 이때 순서는 comp에 의해 결정된다.	first, last, comp
make_heap	범위 first와 last의 밖에 히프를 구성한다.	first, last
make_heap	범위 first와 last의 밖에 히프를 구성한다. 이때 순서는 comp에 의해 결정된다.	first, last, comp
sort_heap	히프 first, last의 원소들을 정렬한다.	first, last
sort_heap	히프 first, last의 원소들을 정렬한다. 이때 순서는 comp에 의해 결정된다.	first, last, comp
min	두 객체중 작은것을 돌려준다. 이때 순서는 comp에 의해 결정된다.	a, b
min	두 객체중 작은것을 돌려준다.	a, b, comp
max	두 객체중 큰것을 돌려준다. 이때 순서는 comp에 의해 결정된다.	a, b
max	두 객체중 큰것을 돌려준다.	a, b, comp
max_element	범위에서 가장 큰 객체에로의 반복자를 돌려준다.	first, last
max_element	범위에서 가장 큰 객체에로의 반복자를 돌려준다. 이때 순서는 comp에 의해 결정된다.	first, last, comp
min_element	범위에서 가장 작은 객체에로의 반복자를 돌려준다.	first, last
min_element	범위에서 가장 작은 객체에로의 반복자를 돌려준다. 이때 순서는 comp에 의해 결정된다.	first, last, comp
lexicographical_compare	범위1의 렬이 자모순으로 범위2의 렬보다 앞서면 참을 돌려준다.	first1, last1, first2, last2
lexicographical_compare	범위1의 렬이 자모순으로 범위2의 렬보다 앞서면 참을 돌려준다. 이때 순서는 comp에 의해 결정된다.	first1, last1, first2, last2, comp
일반화된 수값조작		
accumulate	범위의 매개 객체에 init = init + *iter를 차례로 적용한다.	first, last, init
accumulate	범위의 매개 객체에 init = op(init, *iter)를 차례로 적용한다.	first, last, init, op
inner_product	범위1과 범위2로부터 대응하는 값들에 init = init + (*iter1) * (*iter2)를 차례로 적용한다.	first1, last1, first2, init

이 름	목 적	인 수
inner_product	범위1과 범위2로부터 대응하는 값들에 init = op1(init, op2(*iter1, *iter2))를 차례로 적용한다.	first1, last1, first2, init, op
partial_sum	범위1의 선두로부터 현재반복자까지 값들을 더하고 범위2의 대응하는 반복자에 합을 배치한다. *iter2 = sun(*first1, *(first1+1), *(first1+2), …, *iter)	first1, last1, first2

partial_sum	범위1의 first1과 현재반복자사이의 객체들에 op를 차례로 적용하고 범위2의 대응하는 반복자에 결과를 배치한다.
	answer = *first;
	for(iter=first1+1; iter != iter1; iter++)
	op(answer, *iter);
	*iter2 = answer;
		first1, last1, first2, op

adjacent_ difference	범위1의 린접객체들을 덜고 범위2에 차를 배치한다. *iter2 = *(iter1+1) - *iter1;	first1, last1, first2
adjacent_ difference	범위1의 린접객체들에 op를 차례로 적용하고 범위2에 결과를 배치한다. *iter2 = *(iter1+1) - *iter1;	first1, last1, first2, op

2. 성원함수

서로 다른 용기들에서 류사한 목적을 가지는 성원함수들은 같은 이름을 사용한다. 그러나 어떤 용기클라스나 유효한 성원함수들을 모두 포함하는것은 아니다. 다음의 표는 매개 용기에 유효한 성원함수들을 보여준다.

성원함수	vector	list	deque	multi set	set	multi map	map	stack	queue	priority_ queue
operator==	○	○	○	○	○	○	○	○	○	
operator!=	○	○	○	○	○	○	○	○	○	
operator<	○	○	○	○	○	○	○	○	○	
operator>	○	○	○	○	○	○	○	○	○	
operator<=	○	○	○	○	○	○	○	○	○	
operator>=	○	○	○	○	○	○	○	○	○	
operator=	○	○	○							
operator[]	○		○			○				
operator*		○	○							
operator->		○	○							
operator()				○	○	○				
operator+			○							
operator-			○							
operator++		○	○							
operator--		○	○							
operator+=			○							

성원함수	vector	list	deque	multi set	set	multi map	map	stack	queue	priority_ queue
operator-=			○							
begin	○	○	○	○	○	○	○			
end	○	○	○	○	○	○	○			
rbegin	○	○	○	○	○	○	○			
rend	○	○	○	○	○	○	○			
empty	○	○	○	○	○	○	○	○	○	○
size	○	○	○	○	○	○	○	○	○	○
max_size	○	○	○	○	○	○	○			
front	○	○	○						○	
back	○	○	○						○	
push_front		○	○							
push_back	○	○	○							
pop_front		○	○							
pop_back	○	○	○							
swap	○	○	○	○	○	○	○			
insert	○	○	○	○	○	○	○			
erase	○	○	○	○	○	○	○			
find				○	○	○	○			
count				○	○	○	○			
lower_bound				○	○	○	○			
upper_bound				○	○	○	○			
equal_range				○	○	○	○			
top								○		○
push								○	○	○
pop								○	○	○
capacity	○									
reserve	○									
splice		○								
remove		○								
unique		○								
merge		○								
reverse		○								
sort		○								

3. 반복자

다음의 표는 각 알고리듬에 요구되는 반복자의 형을 보여준다.

알고리듬	입력	출력	정방향	량방향	직접호출
for_each	○				

알고리듬	입력	출력	정방향	량방향	직접호출
find	○				
find_if	○				
adjacent_find	○				
count	○				
count_if	○				
mismatch	○				
equal	○				
search			○		
copy	○	○			
copy_backward	○	○			
iter_swap		○	○		
swap_ranges					
transform	○	○			
replace			○		
replace_if			○		
replace_copy	○	○			
fill			○		
fill_n		○			
generate			○		
generate_n		○			
remove			○		
remove_copy	○	○			
remove_copy_if	○	○			
unique			○		
unique_copy	○	○			
reverse				○	
reverse_copy		○			
rotate			○		
rotate_copy		○	○		
random_shuffle				○	
partition				○	
stable_partition				○	
sort					○
stable_sort					○
partial_sort					○
partial_sort_copy	○			○	
nth_element					○
lower_bound			○		
upper_bound			○		

알고리듬	입력	출력	정방향	량방향	직접호출
eqaul_range			○		
binary_search			○		
merge	○	○			
inplace_merge				○	
includes	○				
set_union	○	○			
set_intersection	○	○			
set_difference	○	○			
set_symmetric_difference	○	○			
push_heap					○
pop_heap					○
make_heap					○
sort_heap					○
max_element	○				
min_element	○				
lexicographical_compare		○			
next_permutation				○	
prev_permutation				○	
accumulate	○				
inner_product	○				
partial_sum	○	○			
adjacent_difference	○	○			

부록 9. 표준 C++의 머리부파일

1. 표준 C++ 머리부파일

표준 C++서고 머리부파일중에는 13개의 STL이 포함되여있다.

머리부파일	목 적	STL
<algorithm>	유용한 알고리듬을 실현하는 형판들을 정의한다.	○
<bitset>	비트모임을 취급하는 형판클라스를 정의한다.	
<cassert>	함수들을 실행할 때 확인을 가능하게 한다.	
<cctype>	문자들을 구분한다.	
<cerrno>	서고함수들에 의하여 알려지는 오유코드들을 시험한다.	
<cfloat>	류동소수점수형속성을 시험한다.	
<ciso646>	ISO 646 가변문자모임으로 프로그람을 작성하게 한다.	
<climits>	옹근수형속성을 시험한다.	
<clocale>	여러가지 문화관례에 적응하게 한다.	
<cmath>	일반수학함수를 계산한다.	
<complex>	복소수연산을 지원하는 형판클라스를 정의한다.	
<csetjmp>	대역이행명령문을 실행한다.	
<csignal>	각종 례외조건들을 조종한다.	
<cstdarg>	가변개수의 인수들을 호출한다.	
<cstddef>	몇가지 유용한 형과 마크로들을 정의한다.	
<cstdio>	입출력을 처리한다.	
<cstdlib>	여러가지 연산을 처리한다.	
<cstring>	여러가지 종류의 문자렬을 조작한다.	
<ctime>	여러가지 시간과 날자형식 사이를 변환한다.	
<cwchar>	폭문자스트림과 여러가지 종류의 문자렬을 조작한다.	
<cwctype>	폭문자를 구별한다.	
<deque>	deque용기를 실현하는 형판클라스를 정의한다.	○
<exception>	례외조종을 실현하는 함수들을 정의한다.	
<fstream>	외부파일들을 조작하는 몇가지 iostream형판클라스를 정의한다.	
<functional>	<algorithm>과 <numeric>에서 정의된 형판용 술어구성을 방조하는 형판들을 정의한다.	○
<iomanip>	인수를 가지는 여러개의 iostream조작자들을 선언한다.	
<ios>	많은 iostream클라스들의 기초로 봉사하는 형판클라스를 정의한다.	
<iosfwd>	iostream형판클라스들을 정의하기 전에 선언한다.	
<iostream>	표준스트림을 조작하는 iostream객체들을 선언한다.	
<istream>	발취를 수행하는 형판클라스를 정의한다.	
<iterator>	반복자정의와 조작을 방조하는 형판들을 정의한다.	○
<limits>	수값형속성을 시험한다.	

머리부파일	목　　적	STL
<list>	list용기를 실현하는 형판클라스를 정의한다.	○
<locale>	iostream클라스에서와 같이 지역에 고유한 동작을 조종하는 클라스와 형판들을 정의한다.	
<map>	련상용기를 실현하는 형판클라스들을 정의한다.	○
<memory>	용기클라스들에 기억기를 할당하고 해방하는 형판들을 정의한다.	○
<numeric>	유용한 수값함수들을 실현하는 함수들을 정의한다.	○
<ostream>	삽입을 수행하는 형판클라스를 정의한다.	
<queue>	queue를 실현하는 형판클라스를 정의한다.	○
<set>	독특한 원소들을 가지는 련상용기들을 실현하는 형판클라스를 정의한다.	○
<sstream>	문자렬용기들을 조작하는 iostream형판클라스들을 정의한다.	
<stack>	stack용기를 실현하는 형판클라스를 정의한다.	○
<stdexcept>	례외를 알리는 유용한 클라스들을 정의한다.	
<streambuf>	iostream조작을 완충하는 형판클라스를 정의한다.	
<string>	string용기를 실현하는 형판클라스를 정의한다.	
<strstream>	기억기안의 문자렬을 조작하는 iostream클라스들을 정의한다.	
<utility>	일반봉사를 위한 몇가지 형판을 정의한다.	○
<valarray>	값지향배렬을 유지하는 클라스와 형판클라스들을 정의한다.	
<vector>	vector용기를 실현하는 형판클라스를 정의한다.	○

2. 표준 C로부터 계승된 머리부파일

표준 C++서고는 표준 C서고로부터 계승된 18개의 머리부파일들을 포함한다.

머리부파일	목　　적
<assert.h>	함수를 실행할 때 확인을 가능하게 한다.
<ctype.h>	문자를 구별한다.
<errno.h>	서고함수들이 알리는 오유코드들을 시험한다.
<float.h>	류동소수점형속성을 시험한다.
<iso646.h>	ISO 646가변문자모임에서 프로그람을 작성한다.
<limits.h>	옹근수형속성을 검사한다.
<locale.h>	각이한 문화관례에 적응하게 한다.
<math.h>	공동적인 수학함수들을 계산한다.
<setjmp.h>	대역이행명령을 실행한다.
<signal.h>	각종 례외조건들을 조종한다.
<stdarg.h>	가변개수의 인수들을 호출한다.
<stddef.h>	몇가지 유용한 형과 마크로들을 정의한다.
<stdio.h>	입출력을 처리한다.
<stdlib.h>	여러가지 조작을 수행한다.
<string.h>	몇가지 종류의 문자렬들을 조작한다.
<time.h>	각종 시간과 날자형식 사이의 변환을 한다.

머리부파일	목　　적
<wchar.h>	폭스트림과 몇가지 종류의 문자렬을 조작한다.
<wctype.h>	폭문자를 구별한다.

참고문헌

1. 리선화 대학용 《프로그람작성법》 고등교육도서출판사, 주체91(2002).
2. Robert Lafore 《The Waite Group's Object-Oriented Programming in C++》 SAMS, 1999.

이 책은 콤퓨터를 전공으로 하는 교원, 연구사, 대학생들을 위한 참고서이다.

C++프로그람개발법

집 필	한영철, 홍이철	**심 사**	김려철
편 집	차현옥	**교 정**	서금석
장 정	서경애	**콤퓨터편성**	여은정
낸 곳	교육위원회 교육정보쎈터	**인쇄소**	
인 쇄		**발 행**	

교-09-1660 부 값 원